CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

Canada

NL 339 (r. 85/09)

The University of Alberta

A MULTIPROCESSOR ARCHITECTURE FOR REALTIME IMAGE GENERATION

by

Kenneth W. Hruday

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1986

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-32568-2

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Kenneth W. Hruday

TITLE OF THESIS: A Multiprocessor Architecture For Realtime Image Generation

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1986

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .....

Permanent Address:
#202 2703-111B St.
Edmonton, Alberta
Canada T6J 4L9

Dated 14, October 1986

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Multiprocessor Architecture for Realtime Image Generation** submitted by **Kenneth W. Hruday** in partial fulfillment of the requirements for the degree of **Master of Science.**

.....

Supervisor

.....

.....

.....

Date

ABSTRACT

This thesis presents an architecture for realtime generation of images. The architecture uses a rendering algorithm that is a variation of the "A-buffer" scheme proposed by L. Carpenter. This method uses a bitmap approximation of polygon surfaces to perform antialiasing. Modifications to the algorithm include using non-intersecting polygons and edge coherence for fragment production with various simplifications to implement parts of the algorithm in VLSI.

The architecture is based on a modular arbitration scheme, which allows easy construction of multiport memory connected to numerous processors. A key feature of the architecture is its ability to expand by simple insertion of processing cards. This architecture can also be scaled to allow more interleaving and more processing nodes.

A simulation was written to test the algorithm and architecture. Image quality was found comparable to that produced by the Movie.BYU graphics package. Realtime performance can be obtained for "typical" images of less than 2,000 polygons and image resolution of 256-by-256 pixels.

Speedup is the factor most critical to achieving higher performance in the architecture. It was found that linear speedup is obtainable for less than 8 processors per node, after this point fragment storage is unable to cope with production.

The main obstacle to speedup by scaling the system is non-uniform distribution of aliasing. Remedies to this problem are proposed. It is concluded that the algorithm and architecture are promising but need more research to balance workload between various phases of the algorithm and on various resources in the architecture.

Acknowledgements

To badly paraphrase Sir. Issac Newton, "If I have seen farther than most men it is only because I have stood on the shoulders of giants". This small contribution to the field is built upon the works and advice of others. The authors of Movie.BYU, H. Christiansen and M. Stephenson, have provided my thesis with a basis of comparison for my parallel algorithm.

I am also indebted to Martin Dubetz for the use of his modified Movie.BYU software and his IIS frame buffer manipulation programs. He was also instrumental in the early phases of the project by tolerating my frequent questions about Movie.BYU. The picture data used in the experiments was drawn from a number of sources, some of it being Martin's "famous" room scene and Judy McGillis's artistic flower pot.

Robert Lake has taken great delight in finding grammatical errors in this thesis and I am grateful for his editorial skills. Ken Bobey has also contributed his time and talent to the improvement of this thesis.

I indebted for advice and guidance to my supervisor, John Tartar. His experience has guided me through a potential minefield of difficulties in researching and writing this thesis. Last but not least, is my wife, Connie, who has given me moral support in my many hours of need and has supplied excellent editorial comments.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Importance of Research in Computer Graphics	1
1.2. The Image Generation Process	2
1.3. Raster Graphics Systems	5
1.4. Scope of the Thesis	5
1.5. Thesis Overview	6
Chapter 2: Rendering Algorithms and Aliasing	7
2.1. Rendering Algorithms	7
2.1.1. Rendering	9
2.1.2. Z-buffer Algorithm	9
2.2. What is Aliasing?	10
2.3. Prefiltering	12
2.4. A-buffer Algorithm	15
Chapter 3: Review of Parallel Architectures for Image Synthesis	21
3.1. Importance of the Frame Buffer	21
3.2. Different Memory Access Schemes	22
3.3. Processor-per-Pixel Approaches	24
3.4. Processor-Per-Object Approaches (Object-Oriented Processing)	26
3.5. Image Oriented Processing	30
3.6. Other Approaches	34
3.7. Summary	36

Chapter 4: A Multiprocessor Algorithm for Rendering 38

4.1. Factors in Algorithm Design 38

4.1.1. Phase 1 40

4.1.2. Phase 2 41

4.1.2.1. Ready_poly 42

4.1.2.2. Ready_scan 43

4.1.2.3. Make_pix 44

4.1.3. Phase 3 45

4.1.3.1. fragen_init 48

4.1.3.2. make_frag 51

4.1.4. Phase 4 53

Chapter 5: The System Architecture 57

5.1. System Control and Operation 57

5.2. System Overview 58

5.3. System Operation 60

5.3.1. Phase 1 60

5.3.2. Phase 2 60

5.3.3. Phase 3 61

5.3.4. Phase 4 61

5.4. A Pipelined, Modular, Equal-Access Arbitration Scheme 62

5.5. Interface to Processing Nodes and Requirements 65

5.5.1. A-bus Arbitrator 66

5.5.2. B-bus Arbitrator 67

5.6. Processing Nodes 67
5.6.1. The VLSI Processor (Pixel Gun) 68
5.6.1.1. The Microprocessor Interface 69
5.6.1.2. The Buffer-Arbiter Interface 71
5.6.2. A-buffer Structure and Function 72
5.6.2.1. Phase 1 73
5.6.2.2. Phase 2 73
5.6.2.3. Phase 3 74
5.6.2.4. Phase 4 74
5.6.3. Memory Access Patterns of the Algorithm 75
5.7. System Limitations 76
Chapter 6: Simulation Results and Discussion 77
6.1. Speedup with Added Processors 77
6.2. Speedup With System Scaling 85
6.3. System Bottlenecks 95
6.4. Data Dependencies 96
6.5. Summary 99
Chapter 7: Conclusions and Future Research 101
7.1. Conclusions 101
7.2. Extensions and Further Work 103
References 106
A1: Simulation Methodology 109
1.1. The Experimental Design 109

1.1.1. The Simulation 109
1.1.2. MOVIE.BYU 110
1.1.3. Polygon Generator 110
1.1.4. Other Software 111
1.2. The Level of Simulation 111
1.2.1. CPU and Pixel Gun States 112
1.3. The Data 115
1.3.1. Distribution of Complexity 115
1.3.2. Heterogeneity of Polygon Size 116
1.4. Simulation Assumptions and Level of Modeling 116
1.4.1. Simulation Parameters 116
1.4.2. Definition of Statistical Parameters Collected 118
1.4.2.1. Total Number of Pixels Covered: 118
1.4.2.2. Total Number of Whole Pixels On the Screen: 118
1.4.2.3. Total Number of Whole Pixels Stored: 119
1.4.2.4. Total Number of Fragments 119
1.4.2.5. Total Number of Antialiased Pixels On The Screen 119
1.4.2.6. Percentage of Frame Covered: 119
1.4.2.7. Whole Pixels (% of Frame): 119
1.4.2.8. Aliased Pixels (All) (% of frame): 119
1.4.2.9. Aliased Pixels Exposed (% of frame): 120
1.4.2.10. Aliased Pixels (% of Stored Pixels): 120
1.4.2.11. Average Number of Pixels per Edge: 120

1.4.2.12. Average Number of Pixels per Polygon: 120

1.4.2.13. Average Number of Overlaps per Pixel: 120

A2: Sample Simulation Output 121

List of Tables

Table		Page
2.1	Fragment Declaration (Taken from Carpenter[28])	16
2.2	Pixel Struct Declaration (Taken from Carpenter[28])	16
5.1	Microprocessor Interface Pins	69
5.2	Whole Pixel Phase Loading Requirements	70
5.3	Fragment Phase Loading Requirements	71
5.4	Output Buffer Interface Pins	71
6.1	Percent Busy of Performance Indices in Phase 2	82
6.2	Percent Utilization of Phase 2 Resources	86
6.3	Percent Activity For Phase 1 In The 4-by-4 Configuration	95
6.4	Time (millisec.) to Render Asymmetric Polygons	99
A1.1	Simulated CPU States	113
A1.2	Simulated Pixel Gun States	113
A1.3	Data Characteristics of Experimental Data Sets	116

List of Figures

Figure		Page
1.1	Image Generation Process	2
1.2	Generic Raster System (From [23])	5
1.3	Scope of Problem	5
2.1	Rendering of Pixels	7
2.2	Point Sampling	10
2.3	Aliasing	10
2.4	Multiple Overlaps[7]	11
2.5	Sampling with a 5-by-5 filter	12
2.6	Weights For Two Filters	13
2.7	Catmull's Prefiltering Method	13
2.8	Bitmap Representation of a Polygon Corner	16
2.9	Visible Fraction of Front Fragment (after[28])	19
3.1	A Comparison of Standard RAM with Whelan's RAM	23
3.2	Selection of Interior Pixels	25
3.3	Triangle Processor Architecture (After Fussell and Rathi[4])	26
3.4	Weinberg's Rendering Architecture (From [36])	28
3.5	A Rectangle Cell (After Locanthi[2])	29
3.6	Fuchs' and Johnson's Multiprocessor Architecture	30
3.7	A Generalized Image-Oriented Architecture (After [12])	31
3.8	The "EXPERTS" Processing System	33
4.1	Rendering of Whole Pixels	41

4.2 Polygon Initialization 42

4.3 Ready_scan Calculations 43

4.4 Whole Pixel Data Structure 44

4.5 Pixel Fragment Generation 45

4.6 Iterative Clipping Against Pixel Columns and Rows 47

4.7 $\Delta x'$ and $\Delta y'$ 48

4.8 Extrapolation of Polygon Edges 50

4.9 Iterative Line Clipping 51

4.10 Clipping Across a Y Boundary 52

4.11 Fragment Data Structure 52

4.12 Merging Two Common Pixel Fragments 54

4.13 Phase 3 Operations 54

5.1 System Configuration 58

5.2 Bus Arbitration Circuit 62

5.3 Arbitration with the Ring-counter 65

5.4 Processing Card 67

5.5 A-Multiport Buffer 72

6.1 Processors Per Node vs. Time 78

6.2 System Speedup vs. # of Processors 79

6.3 Speedup vs. Processors For Test Set #1 80

6.4 Phase 2 Performance vs. Processors For Test Set #1 81

6.5 Phase 3 Performance vs. Processors For Test Set #1 82

6.6 Phase 4 Performance vs. Processors For Test Set #1 84

6.7 Time vs. Configuration for Random Polygons 85

6.8 System Speedup vs. Configuration For Random Polygons 85

6.9 Time vs. Configuration for The Room Scene 85

6.10 System Speedup vs. Configuration For the Room Scene 85

6.11 Speedup vs. Configuration For The Room (16 Proc. Per Node) 86

6.12 Speedup vs. Configuration For The Split Room 91

6.13 Speedup vs. Configuration For The Split Room (16 Proc. Per Node) 91

6.14 Normal CPU Times Over Fast Times vs. Processors (4-by-4) 95

6.15 Time Per Partial Result vs. % Aliasing 97

A1.1 Microprocessor State Diagram 112

A1.2 Pixel Gun State Diagram 112

Chapter 1

Introduction

1.1. Importance of Research in Computer Graphics

Today computer graphics is finding application in many diverse fields. Computer aided design, computer aided learning, flight simulation, advertising, and the film industry are all turning to computer graphics to perform many tasks involving considerable effort and time.

It is a testimony to the need of computer graphics that its use continues to grow despite the heavy computational demands required. Consider, for instance, a typical color monitor with a raster of 512-by-512 pixels.† This represents 1/4 of a million pixels, the calculation of each requiring dozens of floating point operations.

Occasionally the screen is 1012-by-1012 or even 4000-by-4000 — this latter case represents 16 million pixels. This is common in the film industry and is usually accompanied by even more calculations per pixel since there is a greater demand for realism. Considering that 1 second of film needs roughly 22 frames, a full length feature film generated with computer graphics would require a staggering number of computations.

Some computer graphics applications require that the calculations be done fast enough to give the illusion of movement on the screen. Using a color monitor, this "realtime" requirement dictates that 30 new frames of pixels be calculated per second. For the case of 512-by-512 pixels, assuming a meager ten floating point operations per pixel, one needs a computational bandwidth of at least 75 million floating point operations per second. The heavy computational burden posed by computer graphics, coupled with realtime requirements, shows that most current computers are inadequate.

† A "pixel" is a spot of color on the video screen. A computer generated image is simply an array of these color spots.

This thesis proposes an architectural solution to part of the problem. Specifically, a multiprocessor architecture is proposed which runs a modified version of Carpenter's "A-buffer" algorithm. This architecture is modularly expandable and can be scaled to different configurations. This thesis investigates its performance with varying data characteristics and with factors determining achieved speedup. Before defining the scope of the thesis a brief sketch of the image generation process is presented to place it in perspective.

1.2. The Image Generation Process

The following discussion limits itself to a "typical"† approach taken to producing a computer image and therefore excludes techniques such as ray tracing or texture mapping. Applications for typical images range from flight simulation to cartoon animation. Figure 1.1 illustrates the generic process of producing a computer generated image. The reader is refered to the following sources for a more detailed discussion: [19], [40], [23], [41] and [22].

The first step is creating a geometric approximation of the desired scene. Surfaces of objects, such as vases and spheres, are modeled with a polygon mesh. Each polygon in the mesh bounds a region which takes the color of the defining polygon.‡ The next phase consists of transforming the polygon coordinates from object coordinate space to image coordinate space. A scene is usually modeled in a coordinate system convenient for defining and manipulating objects. This system, however, cannot be used for image display. After transformation, the polygon coordinates are in "image" or video-screen coordinate-space.

---

† A "typical" approach uses only polygons for modeling and Gouraud shading for the lighting model. Typical rendering is done with a scanline algorithm. The Movie.BYU graphics package is what this thesis defines as typical.

‡ Note that not all scene modeling uses polygons as the "fundamental" geometric primitive. Curved surfaces such as "bicubic patches" are used for more accurate modeling. Unfortunately, manipulation of bicubic patches is much more time consuming than that of planar polygons. This problem is the primary reason polygons are most commonly used.

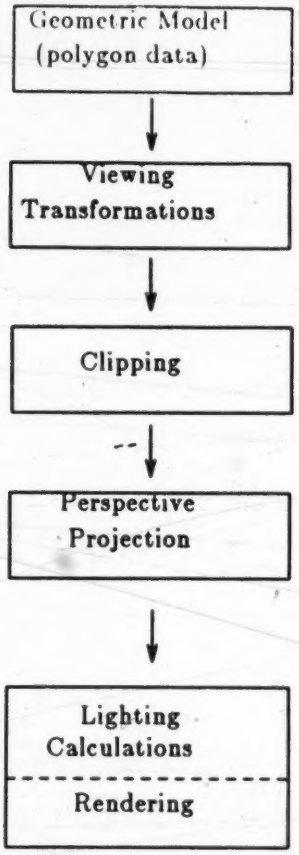


Figure 1.1 Image Generation Process

Clipping is the third step and it eliminates parts of the polygon model that can not be seen in the final image. Perspective projection follows clipping and is necessary because there is no perspective in the polygon model. This "shrinks" the lines and polygons in the image that are most distant from the observation point and gives the appearance of perspective.

All the above transformations can be accomplished with matrix-vector multiplication for which much special purpose hardware exists. There is even a special-purpose graphics chip specifically designed to handle all these operations [25].

The volume of data in the described operations is considerably less than that of

the later phases of image creation. A single polygon could, for instance, be defined with 3 or 4 coordinates. This same polygon could cover an entire video screen and cause production of 262,144 pixels (in a 512-by-512 pixel resolution screen). Clearly, this later part of image creation merits more attention.

After performing the above transformations, the polygons can be converted into pixels on the computer screen. First, lighting of the polygons is simulated with a lighting model. This model is an equation relating the angle of the polygon surface (with respect to the viewer and the light source) to the intensity of light reflected from its surface. A commonly used model is *Gouraud Shading* and the interested reader is directed to the previously cited references or to the original paper for details [20].

The lighting calculations insure that all the polygons have colors associated with their defining points. These colors are represented by three integers corresponding to the red, blue and green components of the CRT display. In good quality displays, these components can take up to 256 values. Thus it is possible to display up to 256^3 different colors.

Algorithms handling the final phases vary greatly but generally they accomplish three tasks before the process is complete. First "hidden surface removal" is performed. This procedure insures that objects (or parts of objects) will not be displayed if they are covered by other objects. Next is a step called "antialiasing," which is discussed and defined in the next chapter. It should be pointed out that antialiasing is computationally expensive so it is often omitted despite the immense difference it makes in the quality of the final image. The last procedure is called "scan conversion" and it produces finished pixels from the polygon primitives.

1.3. Raster Graphics Systems

Figure 1.2 illustrates a "generic" raster graphics system. The image creation system is usually a general purpose computer which performs the modeling and scan conversion of an image, reducing it to a set of integers representing color values for each pixel. These pixel colors are stored into special-purpose memory called a "refresh" or "frame" buffer.

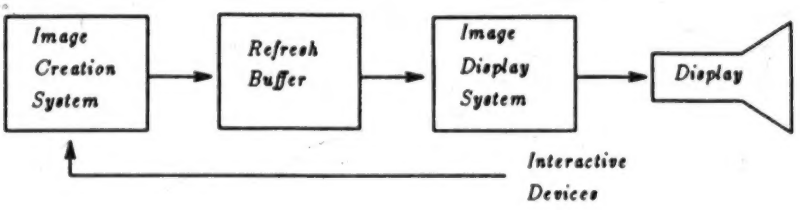


Figure 1.2 Generic Raster System (From [23])

The image display system reads values out of the frame buffer and interprets them as intensities for the color gun of the video display. For a comprehensive review, consult [23]. A more thorough review and description of raster graphics technology can be found in [33] and [26].

1.4. Scope of the Thesis

This thesis proposes an architecture for realtime image production. Specifically, polygon rendering is separated from the image creation system and integrated in a system that contains the frame buffer. Figure 1.3 illustrates the scope of this thesis. Here a host computer performs all the modeling, lighting and transformation tasks, while the proposed architecture and algorithm (enclosed in the dotted box) handles rendering.

Input to this rendering system consists of Gouraud shaded polygons having color values at each vertex. It also assumes that the polygons are convex, non-intersecting, and have been scaled, translated and clipped to the CRT (cathode ray tube) screen

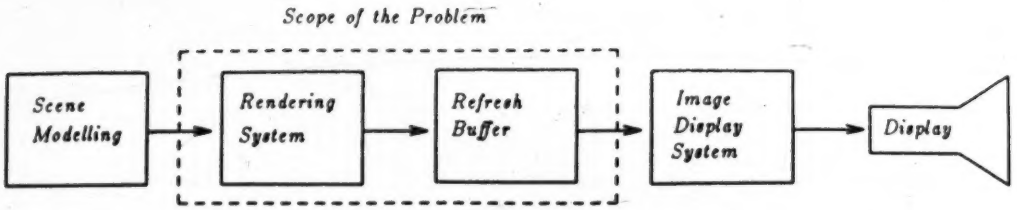


Figure 1.3 Scope of Problem

coordinate system. The proposed architecture performs hidden surface removal, antialiasing and scan conversion to produce a frame buffer of pixels. While lighting calculations could be performed in the architecture, it is felt that these calculations are more properly part of the modeling aspects of scene generation so are beyond the purview of the thesis.

1.5. Thesis Overview

Chapter 2 briefly reviews some concepts fundamental to graphics algorithms and introduces the reader to the *A-buffer* algorithm. The more knowledgeable reader can safely skip this chapter as well as Chapter 3 (which surveys graphics architectures in the literature).

Chapter 4 discusses the algorithms designed for this thesis and is followed by an explanation of the structure and function of the proposed architecture. Results, discussions and conclusions are found in Chapters 6 and 7. This thesis also includes two appendices. The first discusses the assumptions, tools and data used for the experiments. The second appendix is a sample of the simulation output and it includes a summary of all the parameters used in the simulator.

Chapter 2

Rendering Algorithms and Aliasing

2.1. Rendering Algorithms

Given a collection of polygons with color descriptions at each node, how does one produce an image on the video screen? This process is called rendering and consists of producing pixels for the frame buffer (which represent dots of light on the screen). Conceptually, one can view the frame buffer as an n-by-m grid of squares — with each square representing a pixel. Rendering is then the process of determining the color for each square in the grid. This color corresponds to the weighted color value of all the polygons intersecting that square. Consider, for instance, the two polygons in Figure 2.1. When a pixel falls within the polygon interior, it assumes 100% of the polygon's color, otherwise the final color may come from a number of sources including the background. An additional consideration is determining whether objects within the image are covered by other objects (hidden surface removal).

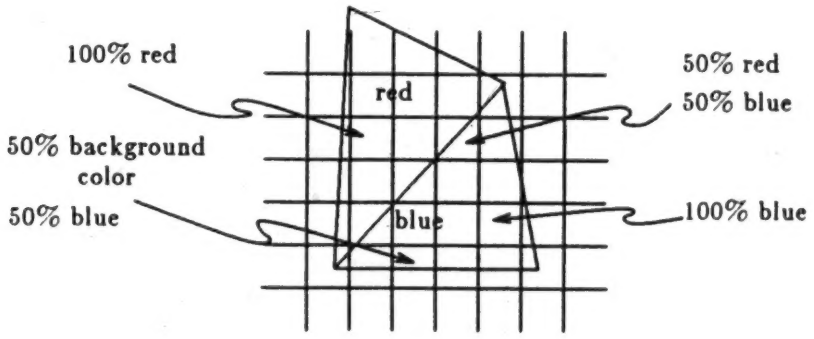


Figure 2.1 Rendering of Pixels

There are many ways of producing pixels. One could, for instance, consider each pixel and check for objects that intersect its area. Another method is to consider each polygon and produce all the pixels that it covers. Yet another possible solution is to consider the image scanline by scanline. All these approaches differ in their degree of

practicality since some (like the pixel approach) make a very inefficient implementation in software, but still make a good hardware algorithm.

Rendering algorithms can be classified according to the approach they take. Two broad categories — not mutually exclusive — are whether calculations are carried out in "image-space" or "object-space."

Image space algorithms calculate a color intensity for pixels on the screen to the limits of screen resolution. Object-space algorithms calculate an image to the limits of computer precision. Thus, images produced in object-space appear correct when expanded many times, while image-space images degrade.

Two elements of rendering algorithms were observed by Sutherland, Sproull, and Schumacker [21,22] to be fundamental in efficient rendering algorithms. Sorting or organizing polygons allows efficient consideration of only those polygons relevant to a particular area of the screen.

The other element, data coherency, is a property of the data indicating that data in one region of space (usually) differs little from data of an adjacent region. This can be used to advantage since it can reduce the computational effort required to render the image.

For a comprehensive review of rendering algorithms, the reader is refered to [22]. More up-to-date coverage can be found in [40], [23] and [41]. The remainder of this chapter considers hidden surface removal and introduces the reader to the concept of aliasing in computer graphics. Various solutions to aliasing are considered and the chapter concludes with a discussion of the A-buffer algorithm.

2.1.1. Rendering

Rendering, or scan conversion is a simple process despite the great amount of time it consumes. The idea is to interpolate pixel colors inside a polygon based on the colors of the polygon vertices.† In the simplest case of rendering a single polygon, one calculates colors along the polygon edges for each scan line. This produces a starting color value and an ending value for the "run" of pixels bounded by the two polygon edges. The pixel colors on a scanline are then linearly interpolated at each pixel center and stored in the frame buffer at the appropriate pixel addresses. This produces shaded pixels.

A simpler approach is to assign a single color to all the pixels in the polygon. The problem is then reduced to determining the pixels covered by the polygon. This solution, however, does not produce realistic images. The rendering problem is made much more complex when more than one polygon is considered, since there is a possibility of overlap or intersection of the polygons. This problem is handled by a hidden-surface algorithm.

2.1.2. Z-buffer Algorithm

This algorithm is perhaps the simplest means of hidden surface removal. The idea is to assign a depth value for every pixel produced. This depth is stored along with pixel color values and permits conditional replacement of pixels in the frame buffer. If, for instance, the resident pixel is closer to the viewer than the in-coming pixel, then the in-coming pixel is rejected. Otherwise the new pixel and depth value replace the old pixel.

An important consequence of this algorithm is that no depth sorting is necessary. This has repercussions for a parallel implementation of the algorithm. Without

† Since a vertex "color" is composed of three color components (red, blue, green), this is actually an interpolation of three different values.

sorting, the data can be distributed among several independent processors. The simplicity of the algorithm is also a major consideration for direct hardware implementation. For these reasons the Z-buffer algorithm is the most common element of recently proposed parallel architectures. Unfortunately, it succumbs to aliasing problems.

2.2. What is Aliasing?

Aliasing, in the context of computer graphics, is most commonly manifested in the appearance of jagged edges instead of smooth ones. This should be recognizable to anyone who has seen cheap video graphics. Aliasing arises from the way that pixel colors are calculated.

Consider the pixels in Figure 2.2. Here polygons intersect the boundaries of both pixels. A rendering algorithm must decide what color to assign to each pixel.

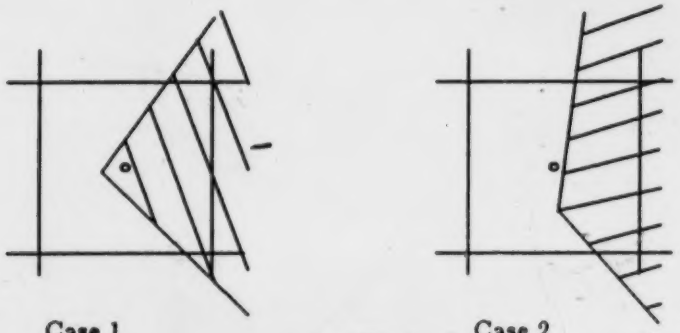


Figure 2.2 Point Sampling

The most common approach in the past was to calculate the color at a single point at the center of the pixel. This has the advantage of being very fast and, in most cases is accurate. With this method the pixels in Figure 2.2 are assigned two vastly different colors since the polygon intersects the pixel center in Case 1 but not in 2. Figure 2.3 illustrates what happens on a large scale. If one were considering an object against the background, the problem would be difficult enough, but when multiple objects intersect a single pixel it gets even more complex. Consider the case taken from

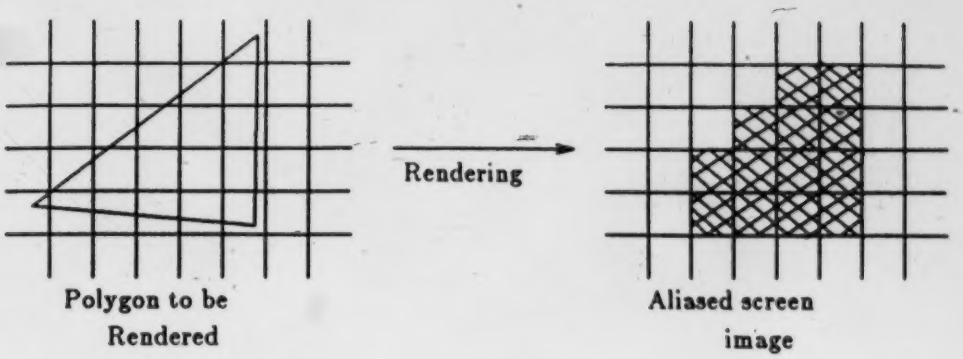


Figure 2.3 Aliasing

Catmull [7], shown in Figure 2.4. Not only must one determine color contributions from each object, but their depth relationships must be accounted for.

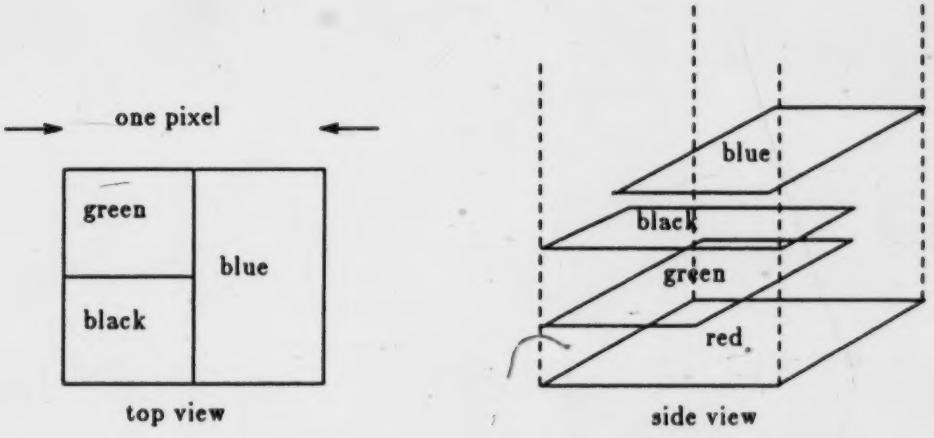


Figure 2.4 Multiple Overlaps[7]

As Catmull points out in [7], correct antialiasing requires some sort of hidden surface algorithm at each pixel. Antialiasing can be a very expensive technique to use since it greatly increases the amount of computation. In fact, Catmull reports that in antialiasing complex images, the total execution time is increased by 3 times using his method.

The cost of antialiasing has led researchers to investigate alternate means of reducing the aliasing problem. Crow has researched the area and compared 3 different

methods in [11]. He also proposes a prefiltering method in [8].

2.3. Prefiltering

There are two approaches to solving the aliasing problem. One can "prefilter" the pixel by calculating the visible object areas as fractions of the total pixel area and use these as weights on the object colors. Alternatively, one can sample within the pixel at many points (*oversampling*) to derive an estimate of the total pixel color. This latter method corresponds to "filtering" the pixel and various filter types can be used.

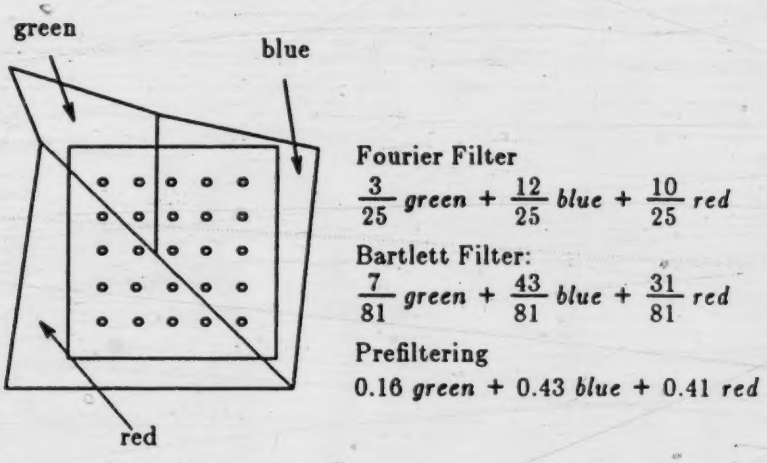


Figure 2.5 Sampling with a 5-by-5 filter

Figure 2.5 illustrates the sample points of a 5-by-5 filter in a covered pixel. For the purposes of discussion, assume that the 3 polygons intersect the pixel are 100% of the primary color indicated. Hence, if the red polygon covers half the pixel, half of the calculated pixel color should be red. This would be the result given by prefiltering.

Figure 2.6 shows the two filters used for the above calculations. The Fourier filter gives each sample point equal weight in the final pixel color while the Bartlett filter gives greater weight to sample points near the pixel center. Theory indicates that the Bartlett filter is better but Crow found that it requires more computation than a simple Fourier filter. Both filtering methods are slower than prefiltering when images of

comparable quality are rendered.

It is instructive to examine a prefiltering method in more detail. The following is a description of the pixel integrator proposed in Catmull's algorithm [7].

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Fourier Filter

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

Bartlett Filter

Figure 2.6 Weights For Two Filters

Catmull's prefiltering method is illustrated in Figure 2.7. The first step is to clip all the polygons to the pixel boundary. This results in several polygons with clipped edges (a clipped edge is illustrated in the figure with a dashed line). The polygons are depth sorted and an edge from the top-most polygon is used to clip the remaining polygons.

This clipping process produces two polygon lists. One set of polygons is covered by polygon "A" while the other is partially covered by a fragment of polygon "B". The clipped edges are marked as clipped and the top-most polygon of each list is checked to see if all its edges are clipped.

In the list headed by "A" all edges are clipped so now the area of polygon "A" is calculated and used to weight its color. This forms a partial result to be added to the other area-weighted colors calculated later.

The list headed by "B" is then clipped along one of its unclipped edges. This also yields two polygon lists and the same termination criteria is applied as above. So the color contribution of a fragment of polygon "C" is calculated here.

Step 4 finally results in two polygon lists where the top polygons have all their

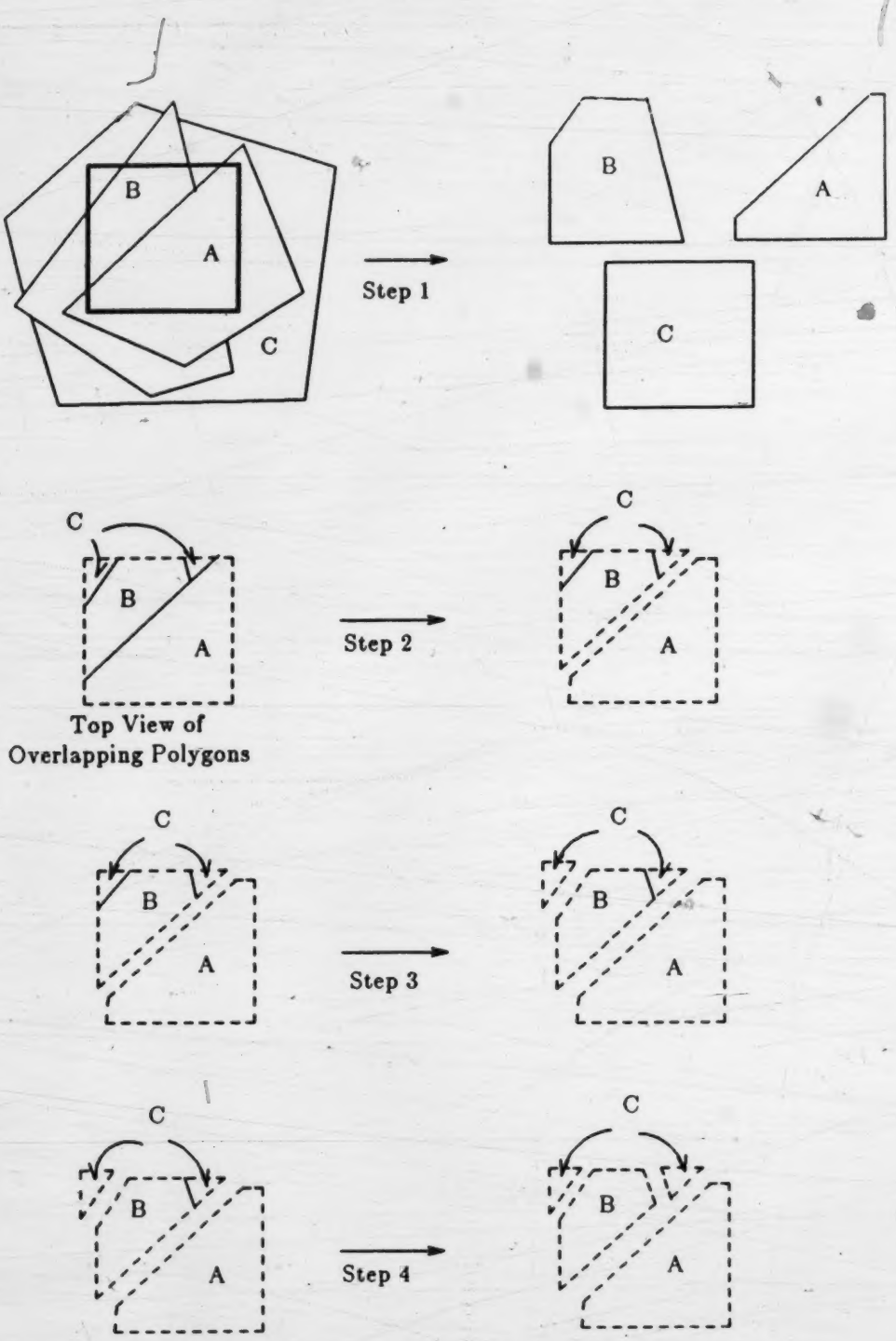


Figure 2.7 Catmull's Prefiltering Method

edges clipped. Clipping now ceases and more partial colors are calculated and summed to give the prefiltered pixel color. The purpose of clipping is to remove parts of polygons not visible in the pixel. Now any area calculations performed on the remaining fragments will only count the visible areas.

2.4. A-buffer Algorithm

There is an interesting approach to the aliasing problem combining elements of prefiltering with filtering techniques. This approach can best be described as a digital approximation to prefiltering. The solution depends on using bitmaps to represent the geometry and area of a polygon within a pixel. This technique was described in 1983 by Fiume, Fournier and Rudolph [6]. A variant of this scheme, called the "A-buffer" was reported the following year by Loren Carpenter [28].

Below is a more detailed explanation of Carpenter's algorithm. Only the antialiasing scheme is described since rendering of "normal" pixels is done by a conventional Z-buffer type algorithm. The algorithm has been modified so that the produced pixels carry two Z values; Z_{min} and Z_{max}, whose use will be explained later.

Bitmaps describing the geometry and area of a polygon within a pixel can be constructed very quickly using "table-lookup". This is done by extrapolating each polygon edge to the pixel boundaries. The resulting intercepts are rounded to a specified accuracy and are used as indices into a precalculated table of bitmaps.†

In Carpenter's scheme bitmaps are 8-by-4 bits. This decision allows bitmaps to be conveniently manipulated in a 32 bit register computer. An example of a 5-by-5 bitmap is given in Figure 2.8. Generated bitmaps form part of a data structure defining a *fragment*. A complete *fragment* declaration, coded in C, is given in Table 2.1. This data structure includes the color of the fragment, its area, opacity, bitmap, object tag

† A table lookup scheme for bitmap generation is described in Chapter 4. Carpenter's description is too vague to judge if there is any similarity between his implementation and the one described in this thesis.

and max-min depth values.

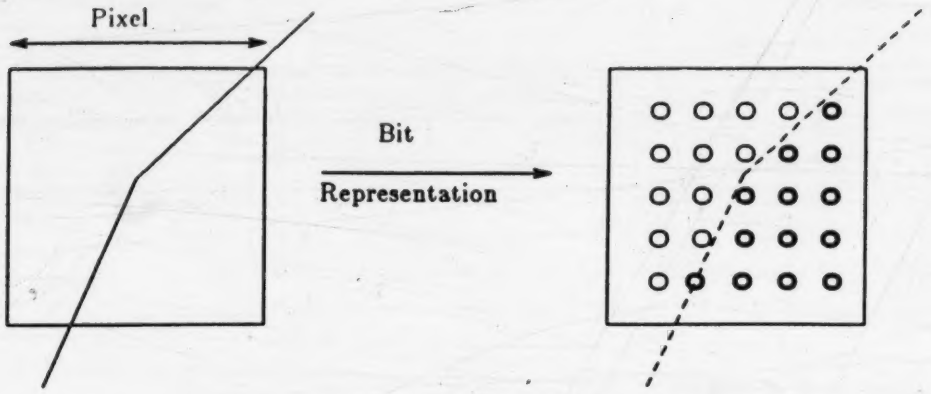


Figure 2.8 Bitmap Representation of a Polygon Corner

The A-buffer also uses a data structure for simple *whole data* representation.† These structures, called "pixel structs" are arranged in an array corresponding to the screen resolution (usually 512-by-512). If the pixel does not require antialiasing, then the Z value in the "pixel struct" is positive

```
fragment_ptr    next;          /* next fragment on list */
short int       r, g, b;       /* color, 12 bits */
short int       opacity;       /* 1 bit transparency */
short int       area;          /* 12 bit precision */
short int       object_tag;    /* from parent surface */
pixel_mask      m;
float           zmax, zmin;    /* positive */
```

Table 2.1 Fragment Declaration (Taken from Carpenter[28])

and the color values for the pixel are given. A negative Z value indicates that the pixel requires antialiasing and the pixel struct contains a pointer to a list of fragment structures sorted by increasing depth. The pixel struct declaration is given in Table 2.2.

Since Carpenter's rendering system uses polygons derived from more sophisticated primitives (such as bicubic patches) it is possible that some polygons could be generated from the same patch in the same pixel. Object tags denote a unique sphere

† *Whole data* refers to data from pixels entirely covered by a polygon.

```
float           Z;      /* negative Z */
fragment_ptr    flist;  /* never null */

                (or)

float   Z           /* positive Z */
byte    r, g, b;    /* color
byte    a;          /* coverage */
```

Table 2.2 Pixel Struct Declaration (Taken from Carpenter[28])

of "patch" and are used to indicate a common origin for polygons intersecting the same pixel. If polygons with the same ID intersect the same pixel, their bitmaps and data structures are combined into *fragment lists*.

The fragment lists are sorted by increasing depth and the packing process follows. *Packing* is a digital version of the "clipping and hidden-surface" algorithm described by Catmull in [7] and is mentioned earlier in this chapter. It starts with a search mask, M_{search}, which defines the visible area remaining in the pixel — initially this is set to all ones. The visible parts of a polygon are defined by the mask, M_{in}. This is, in turn, given by the equation combining the fragment mask (M_f), and the search mask:

$$M_{in} = M_{search} \cap M_f$$

The mask describing the outside area is:

$$M_{out} = M_{search} \cap \overline{M_f}$$

Since the search mask is all ones, all the area of the top fragment is visible. M_{out} now becomes the new search mask of the lower fragments in the list. The recursive calls finish when $M_{out} = \emptyset$. i.e., when all the visible area of the pixel has been accounted for.

The color of a pixel is described by the following equation:

$$C = C_{in} \times A_{in} + C_{out} \times \left(1 - A_{in}\right)$$

where C_{in} is the color in the interior of a fragment, C_{out}, the color of the exterior and

A_{in} the weighted area of the fragment (ranging from zero to one). As in Catmull's method, this algorithm has a simple recursive formulation. The color of C_{out} can be determined by substituting it as C in the above equation. Once done, C_{in} is taken from a fragment found in the area M_{out}. The new M_{out} then becomes the old area without the fragment.

Transparency is also easily handled. The following equation describes the color contribution from a transparent fragment:

$$C_{in} = Opacity_f \times C_f + \left(1 - Opacity_f\right) \times C_{behind}$$

The degree of "opaqueness" is called the opacity factor ($Opacity_f$). This factor determines the transparency of a fragment — a value of "1" means the fragment is completely opaque and nothing can be seen behind it; a value of "0" indicates that the fragment is completely invisible. By this equation, the color of a transparent fragment is the sum of the opacity-weighted fragment color (C_f) and the transparency-weighted color (C_{behind}) behind the fragment. In determining the color behind a transparent fragment, the search mask is simply the current M_{in} value.

Intersections occur in this system and are handled by an approximation. Consider the case given in Figure 2.9. This is a side view of two polygons intersecting within a pixel. The x dimension of the pixel is perpendicular to the plane of page. The visible fraction of the front fragment is be estimated with the following equation.

$$Vis_{front} = \frac{Z_{\max_{next}} - Z_{\min_{front}}}{(Z_{\max} - Z_{\min})_{front} + (Z_{\max} - Z_{\min})_{next}}$$

This fraction is used to weight the colors of the intersecting pieces to give appropriate color blending.

The bitmap solution to aliasing has many benefits. Since it is possible to generate bitmaps with table lookup, bitmaps can be produced quickly. Therefore, at the

expense of storage, bitmaps can be increased in resolution without increasing the time taken in antialiasing.†

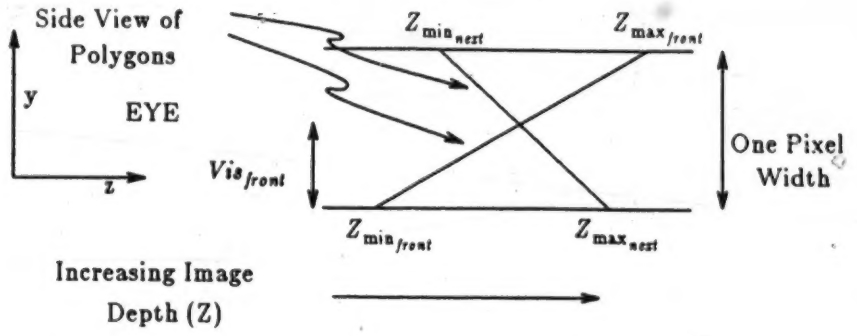


Figure 2.9 Visible Fraction of Front Fragment (after[28])

Clipping and hidden surface removal can be done very quickly since they only involve logical operations. It should therefore be faster than Catmull's scheme. This should also be much faster than filtering methods since only one color is assigned to each bit in a map, whereas oversampling may require a separate color calculation for each point in the sample.

The most significant benefit offered by this approach is that it allows a convenient and efficient parallel solution to the aliasing problem. Past solutions, such as Catmull's algorithm or filtering algorithms, required that all polygon data be present to correctly antialias. This implies global reading of common data and is therefore vulnerable to contention.

In Carpenter's scheme, bitmaps can be produced independently by many processors and stored in some central location to be reconfigured later. Hidden surface removal for normal pixels is handled simply with the Z buffer algorithm which is a good parallel algorithm for hidden surface removal. Since there is no need to sort data in the

† This assumes that the logical operations can be performed in the same time regardless of bitmap size. For bitmap sizes less than the computer register size, this will be true.

Z buffer algorithm, not all the data have to be present at the same time. This algorithm, therefore, could have a pipelined implementation. In Chapter 4, a modification of this algorithm is presented which allows easier implementation in a parallel environment.

Chapter 3

Review of Parallel Architectures for Image Synthesis

Although many architectures have been proposed and implemented specifically for computer graphics, many are designed for either very sophisticated problems such as ray tracing, or much simpler problems such as black and white bit manipulation. This review is restricted to architectures designed for generation of "typical" images which are deemed to be of "moderate" complexity.† In addition, only systems attempting high performance or realtime image synthesis (with purely digital techniques) are considered.

3.1. Importance of the Frame Buffer

Given the enormous bandwidth required in the later phases of image generation, it is not surprising that many of the proposed graphics architectures give memory design a prominent place as part of the solution. The tremendous amount of generated data presents a twofold problem; the first is how to generate the data in real time, the second is how to get it to the frame buffer. We can classify the various approaches on the basis of how they handle these two problems.

A problem related to production is how to distribute the polygons to the processing elements. This is also useful for categorizing architectures. The problem of using pixel data for video signal generation will not be considered here although it does place heavy constraints on the frame buffer design. Whitton [31] gives an excellent tutorial on this issue.

† Typical images are of the quality used for computer aided design, or flight simulation.

3.2. Different Memory Access Schemes

Some papers propose a flexible access scheme to the frame buffer as a partial solution to high speed image synthesis. The principle behind these approaches is that much information stored in the frame buffer is redundant. Also (for some applications) many sequences of operations for one pixel are identical to those performed on others. Thus much time could be saved if one could store and manipulate many pixels simultaneously.

Unfortunately, there is little redundancy in shaded, realistic images, nor is there much need to manipulate pixels individually. So these approaches are best suited to black-and-white, one-bit pixels, or to rendering simple monochrome (unshaded) polygons. It would be instructive, however, to examine two approaches using this strategy.

Bechtolsheim and Baskett [1] have proposed and implemented a system that permits some parallelism in writing and manipulating pixels in the frame buffer. Their system performs frame buffer operations that require five parameters:

$$(X, Y, Width, Operation, Data)$$

where X and Y specify the starting address of a vector of pixels up to width 16. The operation specifies what is to be done with the data and the information at the specified location. All the parameters are updated independently. Their implementation is based on a 1024-by-1024 by one-bit frame buffer intended for use on a microprocessor-based workstation.

A more sophisticated scheme has been proposed by Gupta, Sproull and Sutherland [38] and Sproull et al. [35]. This system, called *The 8 by 8 Display*, permits access to eight-by-eight blocks of pixels. The principal problem that the system overcomes is addressing on arbitrary pixel boundaries. This problem is also found in the previous system and in both cases it consumes most of the hardware and design effort.

The *8 by 8 Display* was implemented for a 1024 by 1024 by one-bit frame buffer using off-the-shelf parts, but a VLSI "smart memory" has also been designed for this architecture. This chip also handles output to the video display. The prototype needs 11 cycles or 2420 nanoseconds to operate on all 64 pixels of an 8-by-8 block. The prototype can paint or erase a 768-by-1024 pixel screen in 93 milliseconds [35].

A promising scheme has been proposed by Whelan [5]. In contrast to the previous two systems which could be implemented with "off the shelf" RAM, this system needs a specially designed RAM. The idea behind the system is simple.

In a typical RAM, one is able to write into a single cell in an array of memory cells. Figure 3.1 shows a single bit of a pixel being addressed at the intersection of the row and column select lines. By a suitable redesign of the basic cell and the selection circuitry, a rectangular region of cells can be selected. The bold lines in the figure indicate the selected lines while the circles indicate selected cells.

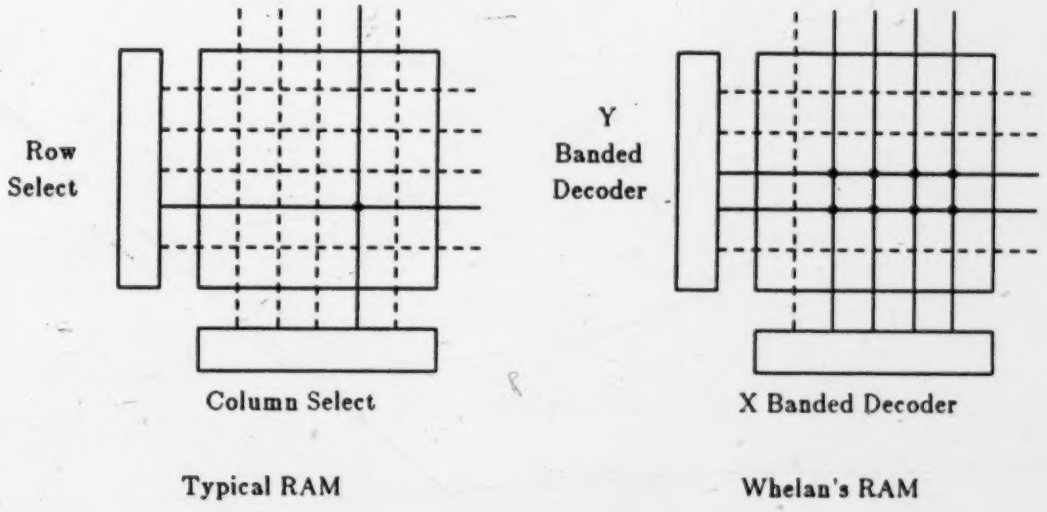


Figure 3.1 A Comparison of Standard RAM with Whelan's RAM

With this redesign it is possible to simultaneously write a single color to all the cells in this rectangular region.

Regions are selected by taking two diagonally opposed points from a rectangle primitive making it possible to fill a whole rectangle in one operation. High bandwidths are possible using this scheme — it is possible to fill the whole screen, for instance, in one memory write cycle.

This architecture is restricted to rectangular geometric primitives and does not permit shading within the rectangle. Shaded antialiased pictures can, however, be rendered by individually addressing pixels having colors different from their neighbors. Since pixel colors in most realistic images are unique, system performance would rapidly degenerate to be (possibly) slower than that of a conventional system. For this reason the system would be extremely useful for VLSI CAD workstations but not useful for general purpose shaded scene generation.

3.3. Processor-per-Pixel Approaches

Recent advances in VLSI technology have made even the most ambitious of architectures seem feasible. A scheme devised by Fuchs et al. [17], for instance, proposes to put a simple processor at every pixel location in the frame buffer. As extravagant as this architecture sounds, it effectively solves the two basic problems of the rendering process. The massive parallelism solves the computational bandwidth problem while communication bandwidth between processors and memory is no longer an issue since data is generated where it's needed.

The system is based on a processor† which effectively calculates functions of the form:

$$f(x,y) = Ax + By + C$$

by broadcasting the coefficients to all the processors in the array. Each element can

† The processors in the array are quite simple. A clever VLSI implementation allows partial results to be distributed to each element so that each element appears to be performing multiplication and addition operations.

then perform color calculations such as:

$$Red = f_r(x,y) = A_r x + B_r y + C_r$$

or depth calculations of the form:

$$Z = f_z(x,y) = A_z x + B_z y + C_z.$$

A simple test equation of the above form can also be used to "set" pixels found in the interior of each polygon. By broadcasting the appropriate A, B and C values for each edge, pixels in the polygon interior yield a positive number while those outside the region give a negative number. Positive pixels are left active and negative ones are set inactive. After broadcasting all the polygon edges, all points in the polygon interior are set (see Figure 3.2).

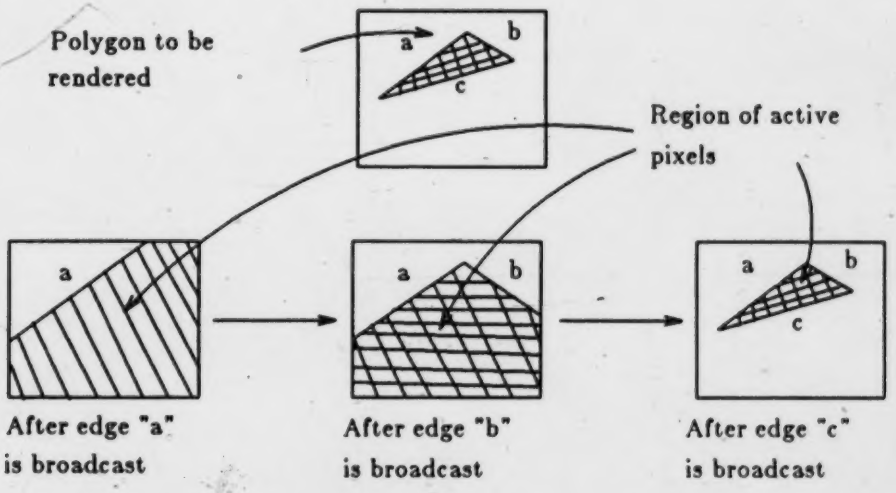


Figure 3.2 Selection of Interior Pixels

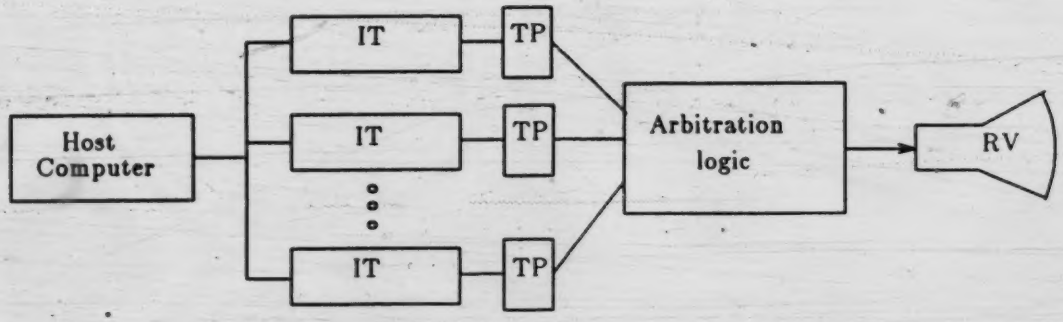
Now all the selected pixels are tested for visibility by calculating depth at each pixel and comparing the Z_{min} value to that already stored there. Pixels covered by previously stored data are disabled and Z is calculated for the remaining active pixels and stored. Color is then calculated in the remaining active pixels by successive broadcasts of the coefficients for the color equations.

The system can handle convex polygons with arbitrary numbers of edges and can perform smooth shading. Unfortunately, the system is incapable of antialiasing. Most recently the system has been implemented in two-micron technology with 64 pixel cells per chip. At this level of integration, a commercial implementation may not be economically feasible, but further advances in VLSI could change this.

3.4. Processor-Per-Object Approaches (Object-Oriented Processing)

There are several papers which allocate a processor to every object or geometric primitive. They can therefore be described as *object-oriented.* These approaches are similar in that all processors generate data for a single pixel or region of pixels at more or less the same time. After data generation, hidden surface removal takes place, usually using the Z buffer algorithm. The following discussion considers two systems similar in concept.

Fussell and Rathi [4], have proposed an architecture which, they claim, should be capable of rendering 25,000 polygons in real time. A simplified schematic of this architecture is given in Figure 3.3.



IT - Interface to host and transformation and clipping
TP - Triangle Processors (1000 per block)
RV - Refresh controller and VDT

Figure 3.3 Triangle Processor Architecture (After Fussell and Rathi[4])

The system is intended to be attached to a host computer which generates clipping

and transformation instructions. Triangles sent from the host to the "IT" units are buffered and the transformation and clipping are performed as specified by the host computer. In addition, the triangles are initialized so they can be processed by the simple triangle processors. Notice that the "IT" units are connected to modules of 1000 processors at a time. This is based on the assumption that the clipping and transformation units can process 1000 polygons in real time.†

The triangle processors and arbitration logic now act as a "virtual frame buffer." This is a "virtual" frame buffer since it does not really hold pixels but the refresh controller operates as if it is reading out of a normal memory. When a read request is sent to the "triangle buffer" all triangle processors generate a pixel‡ and the arbitration logic decides which pixel is closest to the viewer. The arbitration logic now returns this pixel to the refresh controller for display on the video screen. This eliminates the need for a frame buffer.

A desirable feature of this system is that processing time is independent of polygon size and the number of polygons (up to the maximum number allowed). There are, however, a number of problems with this system.

All geometric primitives must be reduced to triangles to accommodate the simple structure of the VLSI processor. Also, since there is one processor per object, the number of processors limits the number of triangles in the scene. The most serious difficulty of this system is the inability to perform antialiasing. The authors admit that adapting the system to perform antialiasing is difficult. This is the greatest problem since an image composed of 25,000 aliased triangles can be easily poorer in quality than an antialiased image of far fewer polygons.

A similar system has been advocated by Richard Weinberg [36,37]. A diagram of

† This is the estimate given for the speed of Clark's geometry Engine which could be used in this unit[4].

‡ Only if the triangle in the processor covers that pixel location.

his architecture is given in Figure 3.4.

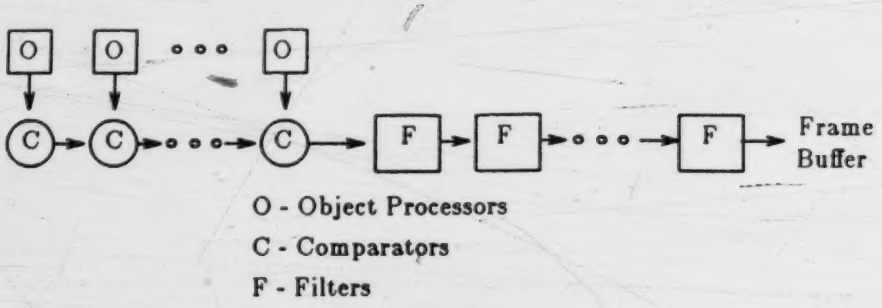


Figure 3.4 Weinberg's Rendering Architecture (From [38])

In this system, object processors receive and render one object per processor and output groups of pixels covered by those objects. The pipelined comparators receive a pixel region for the background colors at the head of the pipeline. At each subsequent comparator stage a check is made to see if the object (stored at the current object processor) covers the current region. If an object partly covers other objects (or is partly visible) then it is added to a list which is passed on through the pipeline. This growing list contains all objects covering the area of current interest.

The filters are also arranged in a pipeline fashion. Each element in the filter pipeline clips a "sub-scanline" from the pixel list and passes the rest of the pixel list on to other filters in the pipeline. Colors are calculated for each sub-scanline and are passed and summed in later stages. The last stage of the pipeline stores the antialiased pixels into the frame buffer.

This method, like the previous one, restricts the number of objects to the total number of object processors. It also requires sophisticated timing and control logic since flow through the pipeline is not uniform.† Weinberg bounds the speed of his system (measured in cycles) with the equation:

$$Number\ of\ cycles = (n \times m) + p + q,$$

† Pixel lists, describing a particular region, could grow or could be partly deleted within the pipeline; this could leave regions of gaps in the data flow.

where n and m are the dimensions of the frame buffer, p is the number of the object processors, and q is the sum of the perimeters of all the trapezoids in the scene.

These two papers are similar in philosophy. Their differences lie primarily in the comparison stages — Weinberg pipelines the comparisons while Fussell and Rathi carry out comparison in a tree-like structure. While Weinberg claims to be able to perform antialiasing, Fussell and Rathi point out that their scheme is also amenable to the same algorithm that Weinberg proposes — they, however, are sceptical about its implementation. It appears that Wienberg's antialiasing scheme may fail under certain "pathological" conditions. Unfortunately, no report was given about any functional or performance simulations that may have been carried out.

A clean, elegant and feasible design is proposed by Locanthi [2]. This system is designed to render rectangles for VLSI design stations. It is restricted to planar, non-shaded rectangles, and does not perform antialiasing or hidden surface removal.

This system is founded on a basic cell illustrated in Figure 3.5. This cell stores two diagonally opposed points of a rectangle. When an X-Y pixel address is broadcast to it, a simple test is made to determine if the pixel falls within the rectangle. If it does, the rectangle color is "ORed" with the colors of other rectangles that contain the pixel. This basic cell is replicated to hold all the rectangles displayed in the scene.

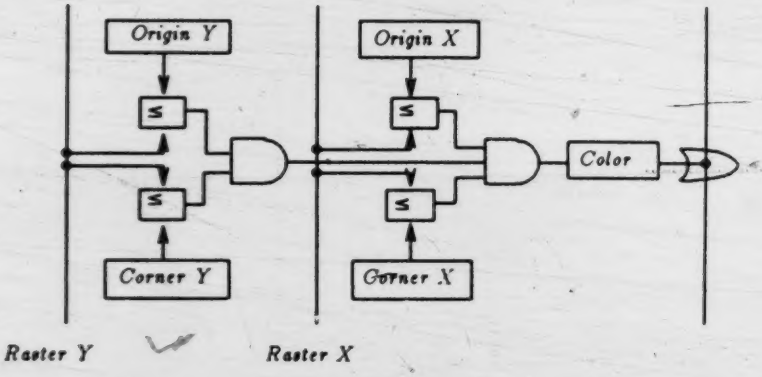


Figure 3.5 A Rectangle Cell (After Locanthi[2])

Although there is no hidden surface removal, this does not necessarily cause a problem with the intended application. As in the previous "object-oriented" designs, the number of objects displayed is limited by the hardware. An advantage with this design is that it eliminates the need for a frame buffer since rendering can occur as the video screen is refreshed. Using technology available in 1979, it seemed feasible to obtain upwards of 200 cells per chip.

3.5. Image Oriented Processing

In contrast to the above approach, it is possible to dedicate processors to particular parts of the image rather than to objects in the image. In these *image-oriented* schemes, processors are responsible for groups of pixels so they are a generalization of the processor-per-pixel schemes. The problem with this solution is that the image may not be uniformly complex so some processors do more work than others.

There are many schemes taking this tack, and they can all be distinguished on how data is distributed to the processors. The first architecture, proposed by Fuchs and Johnson [16], broadcasts polygon descriptions to all processors simultaneously. The structure of the system is shown in Figure 3.6. Notice that all the processors are connected to one or more privately held memory modules.

The purpose of the *CBC* (Central Broadcast Controller) is to broadcast polygon descriptions to all the processors simultaneously. Receiving processors then decide if the polygons intersect their area of the screen. If they do, pixels are generated and stored into the frame buffer. When the last processor has finished with the current polygon, another description is broadcast on the bus.

This architecture is cleverly constructed to allow flexible configuration of memory and processing. It is possible, for instance, to increase (or decrease) screen resolution or to increase processing power by simply inserting or removing memory and processing

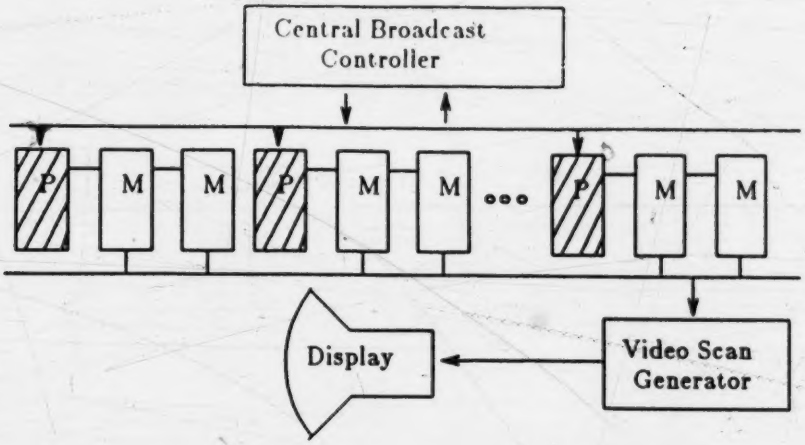


Figure 3.6 Fuchs' and Johnson's Multiprocessor Architecture

cards from the bus. Also, the system is interleaved to allow uniform distribution of the image over all the processing modules. The authors of this paper claim their system is capable of producing a pixel every 50 microseconds.

There are a number of problems with this system. The full parallel potential of the processors can not be realized since the common preparation of the polygons is carried out by all processors. This initialization is necessary before pixels can be produced in the system and it is possible, in principle, to perform them prior to broadcasting.

Another problem is the inefficient use of processors. Since all processors must wait for the slowest one, there is great potential for wasting the computing power of the system. In addition, the broadcasting time is not overlapped with computation so this source of parallelism is also wasted.

Parke [12] generalizes this image-oriented scheme to the system structure in Figure 3.7. Scheme variations are distinguished by their distribution process. Parke compares the broadcast scheme (described above) to a "splitting-tree" and to a hybrid of the two approaches.

Splitting-tree distribution schemes take incoming objects and split them about a

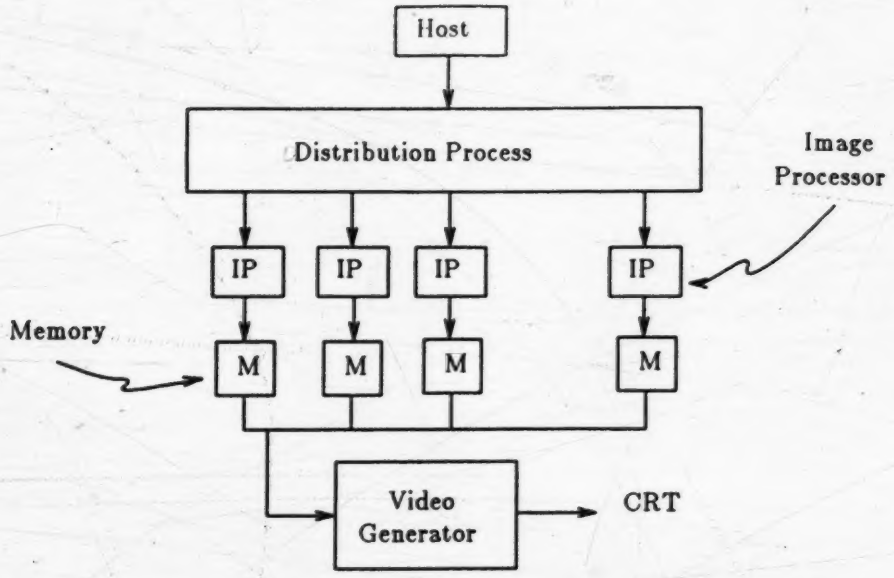


Figure 3.7 A Generalized Image-Oriented Architecture (After [12])

given partition. If the polygons (or polygon fragments) fall on one side of the partition they are sent to one half of the tree, otherwise they are sent to the other half. At each node of the tree — except for the branches — there is a splitter which fragments incoming polygons and sends them to the appropriate processors. This distribution scheme attempts to overcome the problem of waiting for slow processors and achieves more parallelism since I/O and initialization are overlapped with pixel computation.

The performance of this design is vulnerable to the characteristics of the rendered scene. If, for instance, the image is non-uniform in its complexity (a highly probable scenario), some parts of the splitter tree will be overloaded while others remain idle. The splitter tree approach requires that processors be responsible for a contiguous region of the final image, thus there is no interleaving to disperse the workload.

The hybrid approach combines the splitter tree with the broadcaster. In this method, the splitter splits and distributes polygons to two halves of a tree but uses a broadcast approach at the bottom. The hybrid scheme tends to "even-out" the defects

of both approaches.

Still another "image-oriented" design, called "EXPERTS", has been proposed by Niimi et al.[18]. This system incorporates two specially designed, microprogramable processors for rendering — the SLP (scanline processor) and the PXP (pixel processor). These two processing elements are specially designed to implement a scanline rendering algorithm. The system partitions the image space into bundles of scanlines with processors dedicated to rendering them and roughly follows the generic architecture described by Parke (see Figure 3.8). The host computer distributes the polygon descriptions

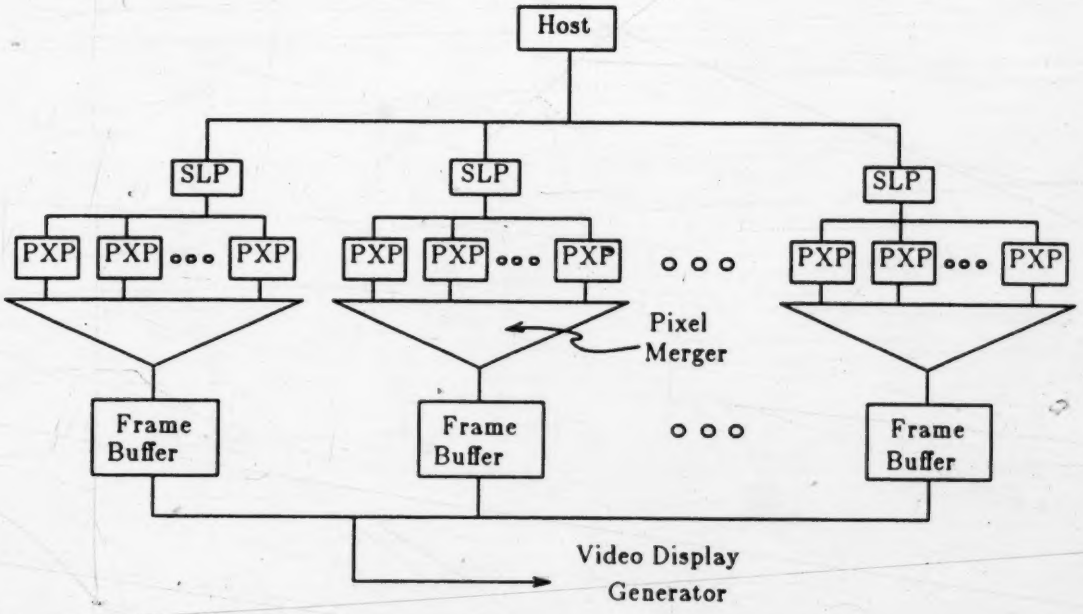


Figure 3.8 The "EXPERTS" Processing System

to the scanline processors which prepares active segment lists. These segment lists are passed on to the pixel processors for pixel production.

The authors claim this system is capable of rendering about 200 polygons in 1/15 of a second. This system is also capable of antialiasing and handling transparency. The authors use an interesting approximation for the antialiasing scheme which also helps

speed system rendering.

3.6. Other Approaches

A general purpose multiprocessor has been considered for use in high speed rendering [6]. The authors of this paper show how high speed graphics can be implemented on a general purpose system called an *Ultracomputer*. This is a parallel processing system being developed at New York University.

The proposed system consists of n independent processing elements connected to n independent shared memory modules. Connection is achieved using Lawrie's Omega network switches which have been augmented with some additional processing capability. The switches have the ability to merge simultaneous operations on any given memory location. Thus, memory traffic can actually be reduced as operations combine in the network.

No performance estimates are given for the approach, although a claim is made that speed-up is linear up to some lower bound defined by the screen resolution. The proposed algorithms have been implemented using the parallel language, Concurrent Euclid.

Crow and Howard [10] have designed a "smart" frame buffer with a view to speeding up picture generation. They use a 32-bit-word frame buffer with a smart update port which can directly implement the Z buffer algorithm, or can combine incoming data with pixels already in the frame buffer. The emphasis in this design is more on flexibility than speed.

Whitted [39] has also suggested a piece of hardware that resides with the frame buffer. He has designed a special LSI chip to produce both pixels and their depth values. This chip is intended to interface with the front end of a frame buffer and accept initialization parameters from the host machine. Performance simulations indicate that using the chip can at least double the speed of a Z buffer algorithm compared

to a VAX 780 alone.

A special bus design has been proposed to aid Z depth comparison [34]. Gemballa and Lindner describe a *multiple-write* bus scheme, where several processors can write to the bus at the same time with the result being the "OR" of all the data output. Writing processors can then check their output against the most significant bit on the bus. If this bit is greater than the corresponding bit in the processor, the processor withdraws its output. Otherwise processors check the next lowest bit, and continue until the lowest order bit is reached and only one processor remains (assuming that all the data is unique).

The authors observe that this scheme can be used to pick the closest pixel among a number of pixels written to the bus. This approach requires that all data going to a particular location be written at once. This implies that the multiple-write bus must be used in an object-oriented architecture.

A scheme involving microprogramming has been proposed by Jackson [24]. This approach involves non-shaded polygons and no antialiasing. Scanlines are run-length encoded by a host processor and are sent to a special purpose processor which can decode and fill the frame buffer at high speed.

Crow [9] describes a scan conversion scheme for vector generation on a raster display device. This plan uses a special bit-slice processor which sorts and maintains vectors and scanlines. As in Jackson's scheme, run-length encoding is used to handle bandwidth problems to the picture element processor. This architecture uses only 2 scanline buffers in place of an entire 512-by-512 pixel frame buffer.

3.7. Summary

Experiments with memory access schemes indicate that they are only good in limited applications since the major issue of pixel generation is not solved. Whelan's scheme in particular, would be ideal for VLSI workstations displaying Manhattan type designs. The vertical and horizontal edges of this application do not have problems with aliasing defects so antialiasing is not important.

Processor-per-pixel approaches — of which there is really only one reported — promise to be very fast and offer shaded images but no antialiasing. But current levels of integration make any usable sized implementation infeasible. This is likely to be the case in the foreseeable future.

Various schemes are object-oriented. These are characterized by having each processor dedicated to rendering one image primitive. A general property of such systems is that processing time is independent of the number of primitives up to a certain limit, namely the number of processors in the system. It does not appear that antialiasing can be accomplished easily, although Weinberg claims his system is capable of it. In the short term, this type of architecture may find only limited application. Locanthi's scheme, for instance, appears feasible and is ideal for VLSI workstations.

Architectures have been proposed where processing power is dedicated to parts of the image rather than to objects. Some of the architectures using this scheme are modular and flexible. The most obvious defect in this approach is its susceptibility to non-uniform complexity of the image. If a complicated image falls in part of the image space, the processors in this region will have more work than the others. Few architectures taking this approach have antialiasing capability.

Other approaches do not appear to come close to the requirements of providing fast, shaded, antialiased images. Most of the other schemes, for instance, are just hardware aids for the Z-buffer algorithm. The Ultracomputer system, provides shading

and antialiasing, but realtime production and display requires special purpose hardware if only to handle the bandwidth of data to be written into the frame buffer.

There are many desirable characteristics that a high performance rendering system should have. Anti-aliasing, with shading and transparency is necessary for realistic image generation. The problem is that this must be done very quickly. In addition, it is desirable for the system to allow flexibility in the type of rendering algorithm used and to be easily changed to give higher speeds. Independence of data complexity and performance, are necessary for a "robust" system. Nearly all the systems examined here have one or more of the above properties but none has them all.

Chapter 4

A Multiprocessor Algorithm for Rendering

4.1. Factors in Algorithm Design

Real-time requirements place severe constraints on any algorithm considered for a system. A limiting factor in most parallel systems is the amount of communication required between processors. The amount of communication required is indicative of the computations "partitionability." Kaplan and Greenberg [29] suggest this is the major consideration in choosing a hidden-surface algorithm for parallel implementation and is given great emphasis in the proposed algorithms.

Other factors considered, are the types of operations performed by the algorithm. Floating point operations, for instance, are time consuming and should be avoided. Even greater speed-up is gained by placing the most frequently executed operations in hardware. But to easily "offload" the algorithm into VLSI, the algorithm must be simple — this means again, no floating point operations.

Fortunately the rendering problem is well bounded with respect to numerical values. Numbers, representing pixel addresses, all fall between 0 and 255, primary color values can be represented by 8 bit integers and a 16-bit integer is sufficient for the Z-depth. All these calculations can therefore be performed with fixed-point arithmetic as the "dynamic range" of numbers is small. These observations prompted the decision to put most of the workload into dedicated hardware. Thus, the pixel generation algorithms are simplified for VLSI implementation. This dedicated VLSI device is called a pixel-gun.

The algorithm presented in this thesis is based on the work of Loren Carpenter [28] and uses two data structures similar to those in his scheme. *Whole pixels* are generated from polygon interiors and are completely covered by a polygon. *Fragments* are pieces of polygons that partially cover a pixel and are represented by structures

containing bitmaps. In the present scheme all fragments are generated from polygon edges.

But Carpenter's algorithm requires some changes and these include data restrictions to allow simplification and changes to take advantage of data coherency in the rendering and fragment generation phases. Simplification is necessary to allow efficient parallel decomposition and to allow some of the algorithm to be performed on a special-purpose VLSI device.

The proposed algorithm assumes non-intersecting polygons so intersecting polygons must be split (if they occur). Splitting has no effect on the appearance of the image but it allows a simpler merging algorithm . This assumption places all aliased pixel locations on polygon edges and allows efficient fragment production since edge coherence can be used to generate bitmaps.†

Fragment generation is also simplified to produce bitmaps of regions on the interior side of the polygon edge. This may lead to an inaccurate representation of the polygon geometry within a single bitmap, but the correct geometry is reconstructed in a later phase of the algorithm. Simplifying this generation makes a pipelined VLSI algorithm much easier to implement.

Whole-pixel and fragment generation have been simplified to the point where both can be performed on the same VLSI device using fixed point arithmetic. Since these aspects of rendering are the most time-consuming, some real performance gains can be expected.

The proposed rendering algorithm is divided into four distinct phases. In the first phase the frame buffer is initialized to the background color. In the second only whole-pixel data are generated, in the third, only fragments and in the fourth the two data types are merged into finished pixels. Partitioning of phases is not strictly

† Polygon splitting may cause more work for the image modeling system.

necessary since both fragments and whole-pixel data can be generated concurrently. Phase partitioning is chosen for the following reasons.

If whole pixels are stored first then fragments can be rejected with only one comparison. In a more concurrent scheme the same fragments could not be rejected if stored before the whole pixel. Since fragment storage is time consuming, unnecessary fragment acceptance would be a detriment to system performance. More fragment storage into the frame buffer puts a greater demand on memory capacity. Also, more fragments will lengthen the merging phase.

Depth comparison would be more time consuming since a whole pixel or a fragment can not be rejected using depth alone. A depth, for instance, can belong to a fragment that does not cover a whole pixel; hence, even if the incoming pixel fragment is behind the resident one, it may still be visible. Therefore, phase partitioning is justified.

This chapter considers the designed algorithms in more detail. Phases 1 and 2 are discussed and three major algorithms, used in Phase 2, are described; these include *ready_poly*, *ready_scan* and *make_piz*. Phase 3 discussion centers on the *fragen_init* and *make_frag* algorithm; it also considers some possibilities for implementing *fragen_init* into VLSI. Phase 4 is considered last and the merging and clipping algorithms are explained.

4.1.1. Phase 1

Phase 1 consists of frame buffer initialization by setting the background color and the depth and dz values.

4.1.2. Phase 2

In Phase 2, "whole pixels" are produced and stored into a frame buffer. As mentioned previously, "whole pixels" are those elements falling in the interior of polygon primitives and have full bit map descriptions (see Figure 4.1).

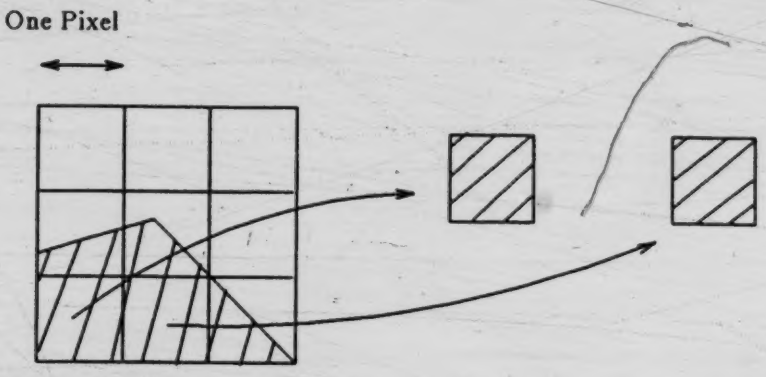


Figure 4.1 Rendering of Whole Pixels

This phase generates pixels in scanline order starting from the top of the polygon primitive (highest y value) to the bottom (lowest y value). Since the pixel-gun performs whole pixel generation, this phase must be simplified to allow VLSI implementation. Unfortunately, the pixel-gun processor is unlikely to be sophisticated enough to handle raw polygon descriptions so it must be initialized before each scanline. Initialization will be done with a more flexible, but slower, general-purpose processor. Polygons also require initialization and in preparing them (using *ready_poly*), incremental parameters are produced for the scanline initialization routine. These include edge colors and coordinate changes ($\frac{\Delta X}{\Delta Y}$, $\frac{\Delta Z}{\Delta Y}$) along each polygon edge. The scanline initialization routine (*ready_scan*) uses these increment values to calculate new edge values. After calculating edge parameters, a new set of increments are produced for the pixel-gun. These routines are now examined in more detail.

4.1.2.1. Ready_poly

This routine is responsible for preparing a new polygon for scanline rendering. When *ready_poly* receives a polygon, it discards horizontal line segments and tries to find the top two edges that bound an area of the polygon. In Figure 4.2 the bold polygon edges bound the top scanlines of the polygon.

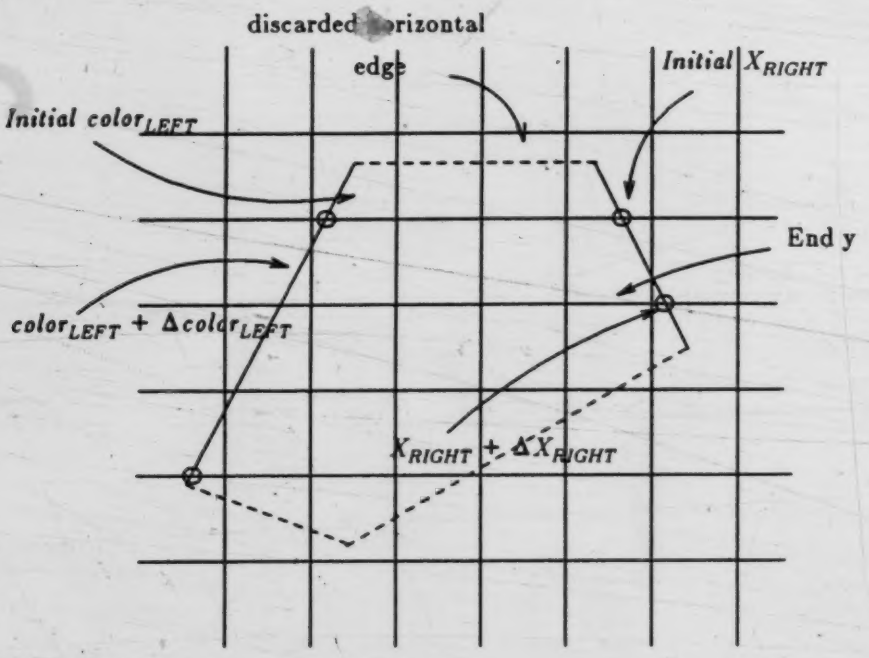


Figure 4.2 Polygon Initialization

Increment values are calculated for both these edges and are added to the current left and right values when a new scanline is started. In this figure, the new x value for the right edge is: $x_{left} + \Delta x_{left}$. Similarly, the new right color is: $color_{right} + \Delta color_{right}$.† Notice, however, that the pixel color is calculated at the center of the pixel while the x value is calculated at the bottom. Discontinuous color changes might result in later rendering phases if the color is incremented on pixel boundaries. The incremented

† Although the diagram indicates color calculation with a single value, three different color components are used.

edge values are: x, z, red, blue and green.

The next event is the determination of when a new polygon line segment needs to be considered. This must be done when the next y value or scanline is not bounded by the current two line segments. In the figure, the "end y" occurs above the bottom of the right line segment. This value is chosen because the right side ends sooner than the left.

If the line segment ends exactly on a pixel boundary (the case with the left line segment) this value will become an end y value since the scanline is still bounded by it. Now *ready_poly* interpolates the colors, z and x coordinates for the first scanline bounded by both line segments. These points are circled in the figure at the top of the polygon. After this, processing is taken over by *ready_scan*.

4.1.2.2. Ready_scan

The first function of this routine is to decrement the current y value. If the new scanline equals the previously calculated "end y" value a new line segment needs to be added, otherwise the function proceeds as follows.

The next step is establishing the beginning and end of the whole pixel "run" on the new scanline. The intersection points of the polygon edge with the boundaries of one scanline are circled in Figure 4.3. The ceiling of the maximum x intersection point on the left edge defines the first possible point of a run of whole pixels (start x). Similarly, the floor of the minimum of the intersections on the right edge defines the lowest x value where a whole pixel run must end (stop x).

If start x is greater than or equal to stop x then there are no whole pixels on the current scanline. A new scanline is started and the process iterates until the current line segments are finished or whole pixels can be found. Whole pixels are indicated in the figure with cross hatching.

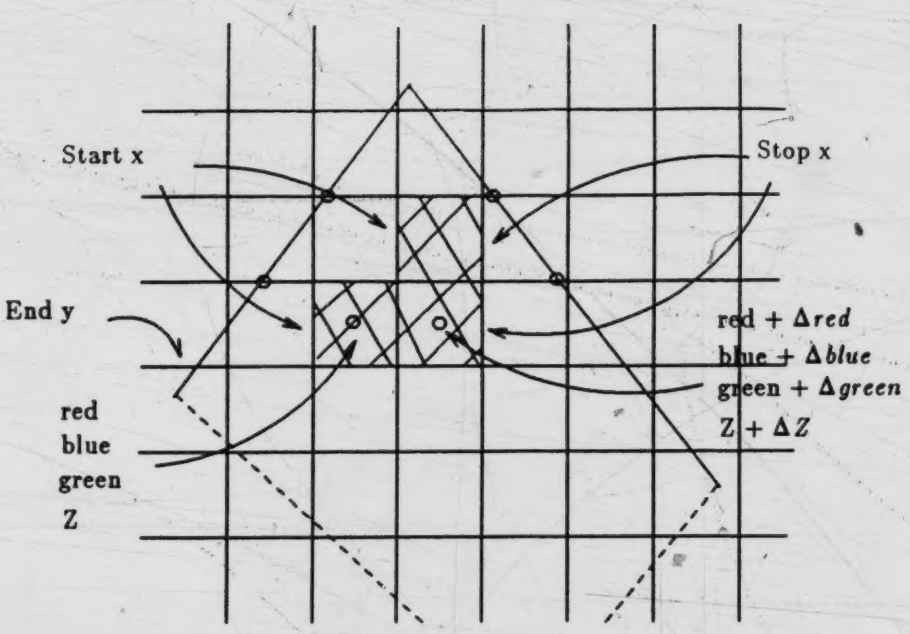


Figure 4.3 Ready_scan Calculations

When the beginning and end of a whole pixel run is found, the new starting values can be interpolated from the polygon edge to "start x." These colors and z depth represent the values for the first whole pixel on the scanline.

The last step in scanline initialization occurs with the calculation of increment values for red, blue, green and z. These are used by the *make_piz* algorithm for calculating new pixels.

4.1.2.3. Make_pix

This algorithm is executed by the pixel-gun. It compares the current x value to the stop x value. If the current x equals stop x, the algorithm stops, otherwise it adds Δ values to calculate the new colors and z. The new pixel is then output, the current x incremented and the process repeats. The pixel produced by this phase is given in Figure 4.4.

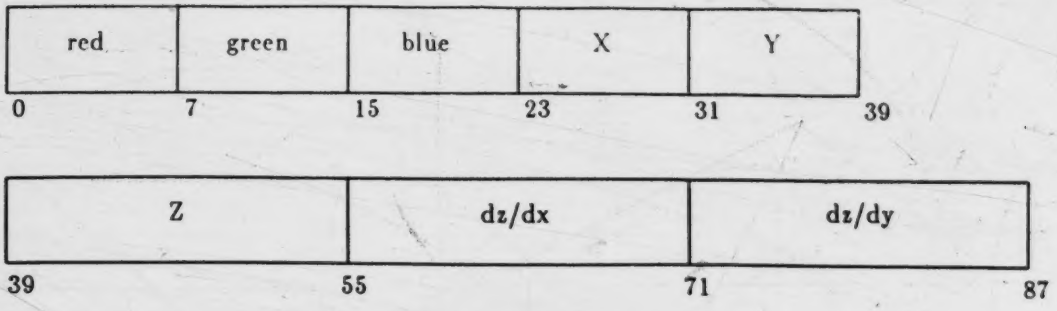


Figure 4.4 Whole Pixel Data Structure

4.1.3. Phase 3

The purpose of Phase 3 is to follow each polygon edge and produce bitmaps of pixels intersecting the edge (see Figure 4.5). Whole pixels are not generated in this phase, although occasionally a full bitmap is produced. Also notice that corners produce two bitmaps even though they fall into one pixel. This is a consequence of the fragment generation algorithm since the pixel-gun only handles one edge at a time and always extrapolates polygon edges to the pixel boundary when producing bitmaps. Polygons narrower than a pixel also produce two bit maps — one map for each opposing edge. In cases such as these, the "true" bitmap of the area is reconstructed in the fourth phase.

To generate fragments, an algorithm must clip the polygon edge to each pixel boundary. This clipping is executed by the VLSI pixel gun and must therefore be fast and simple.

After clipping it is possible to generate a bitmap approximating both the area of the polygon fragment and its geometric information. This allows judgement of whether a particular polygon fragment is covered. Each bit in the map represents 1/16 of the total pixel area and is used in Phase 4 to weight the color value of the fragment for final pixel color determination. The pixel-gun must do bitmap generation as well as

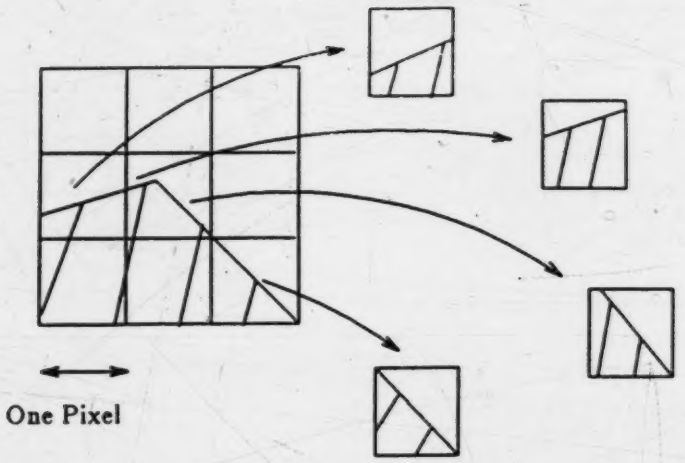


Figure 4.5 Pixel Fragment Generation

produce the red, blue and green colors, the Z depth and the x, y addresses for each pixel fragment.

Color production and depth calculations can be carried out exactly the same way as is done with the whole pixels. Specifically, as the edge is traversed, new depth and color values are calculated by adding increment values. This is important when considering implementation in VLSI because the same circuitry could calculate color and depth for fragments and whole pixels — the only differences between the two phases is in x, y address generation, and generation of a bitmap.

The VLSI based architecture proposed by Fuchs, Poulton, et al. [17] is capable of producing a "pixel map" of all pixels bounded by the edges of any particular polygon. Note that just as all polygon edges define a bounded region of pixels, the pixel boundaries and polygon edges define a bounded region of "sub-pixels"; thus the problem of producing a fragment bitmap is exactly analogous to that of producing a map of pixels.

The pixel planes algorithm can be adopted in a straightforward fashion to bitmap generation. Using this algorithm would be advantageous since it has been extensively

tested by Fuchs et al. in two different implementations — with the second version containing 64 pixels in 2-micron technology. In addition, bitmap generation requires only a subset of this hardware since bitmap size is 16 bits with a depth of one bit, while the "pixel planes" algorithm uses 64 pixels and a depth of 16 bits per pixel. It is therefore feasible, and desirable, to adapt this algorithm to bitmap generation.

An alternate, faster, implementation could use precalculated maps and lookup tables for bitmap generation since only a small number of bitmaps are possible.† The bitmap table contains precalculated bitmaps of polygon interiors based on the assumption that edges are traversed clockwise about the convex polygon. This assumption allows prediction of which bits fall within the polygon interior (see Figure 4.8). Bitmap table size does not need to be large since bitmap symmetry permits a subset of cases to cover all possibilities.

Although table lookup simplifies VLSI implementation it also uses greater chip area. Also, higher speed is not needed since bitmap calculation can be overlapped with output of the other results from the pixel-gun.

Bitmap generation requires pairs of x, y values to determine where the edge enters the pixel and where it leaves — these are the points where the polygon edge is clipped to the pixel boundaries. It was thought that a DDA (digital differential analyzer) algorithm could be adopted to produce the clipped edge values, but unfortunately none of those examined were accurate enough, or simple enough for VLSI implementation. An iterative clipping algorithm was therefore devised to produce the appropriate values.

This algorithm is founded on the following observations. In Figure 4.6 the polygon edge intersects columns of pixels with exactly the same y distance between intersections ($\Delta y'$). The edge also intersects rows of pixels with the same x distance ($\Delta x'$). All the x, y intersection points fall on multiples of these $\Delta x'$ and $\Delta y'$ values

† This technique is used in the experments to simulate the architecture and algorithm.

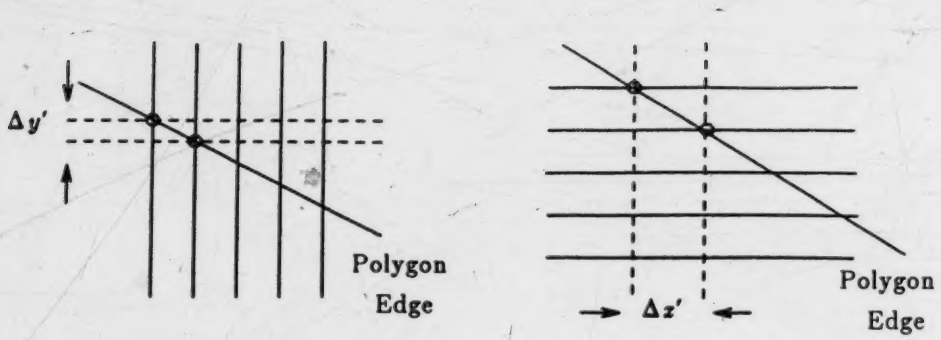


Figure 4.6 Iterative Clipping Against Pixel Columns and Rows

and, in principle, intersections can be calculated by simply incrementing with the Δ values.

Iterative clipping can be performed by deciding which values to add and whether to take the ceiling or floor of x, y addresses. This greatly eases the burden on the VLSI pixel-gun since it avoids multiplication and division, requiring only simple logic and fixed point addition.

The third phase consists primarily of two routines: *fragen_init* and *make_frag*. The function of these routines is elaborated in the next two sections.

4.1.3.1. fragen_init

This routine is responsible for maintaining a list of polygon edges and for determining the edges needing processing. Its primary function, however, is to initialize edges and parameters for subsequent processing by the fragment generation routine — *make_frag*.

Among the initialized parameters are the: Δ colors, Δz, $\Delta x'$, $\Delta y'$, z_{rem} and y_{rem} (the function of these last two parameters are discussed later). The Δ colors (red, blue and green) and Δz value are ratios of the color or z change divided by the total change in y from the top of the polygon segment to the bottom $(y_2 - y_1)$ (Figure 4.7). So Δred is actually: $\frac{red_2 - red_1}{y_2 - y_1}$. This is used to increment the current color whenever the y

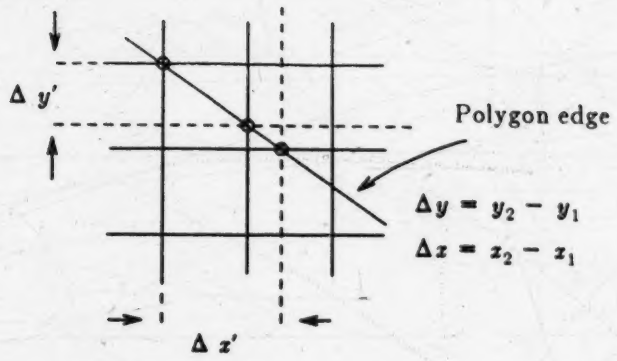


Figure 4.7 $\Delta x'$ and $\Delta y'$.

pixel value is changed. Note that if the polygon segment is horizontal, the x change is used in the ratio.

The $\Delta x'$ and $\Delta y'$ values are illustrated in Figure 4.7. $\Delta y'$ is similar to the slope of the polygon edge, while $\Delta x'$ is similar to the inverse of the slope. $\Delta y'$ is defined as:

$$\Delta y' = \begin{cases} \Delta y/\Delta x & \text{if } \Delta x \textit{ is positive} \\ -\Delta y/\Delta x & \text{if } \Delta x \textit{ is negative} \end{cases}$$

Similarly, $\Delta x'$ is defined as:

$$\Delta x' = \begin{cases} \Delta x/\Delta y & \text{if } \Delta y \textit{ is positive} \\ -\Delta x/\Delta y & \text{if } \Delta y \textit{ is negative} \end{cases}$$

These values are fundamental to the *make_frag* algorithm and their use is described later.

The next major function of this routine is to interpolate the start of the polygon edge back to the pixel boundaries. *Make_frag* assumes, for the sake of simplicity, that all the polygon edges start on pixel boundaries. It is important to note that the polygon edges are assumed to have direction. This assumption is used in generating bitmaps in a later phase, consequently all polygon edges are traversed in the clockwise direction. This means that the first point in the edge is considered to be the starting point and is extrapolated back to the pixel boundary. In Figure 4.8 this extrapolation

is indicated by a dashed line.

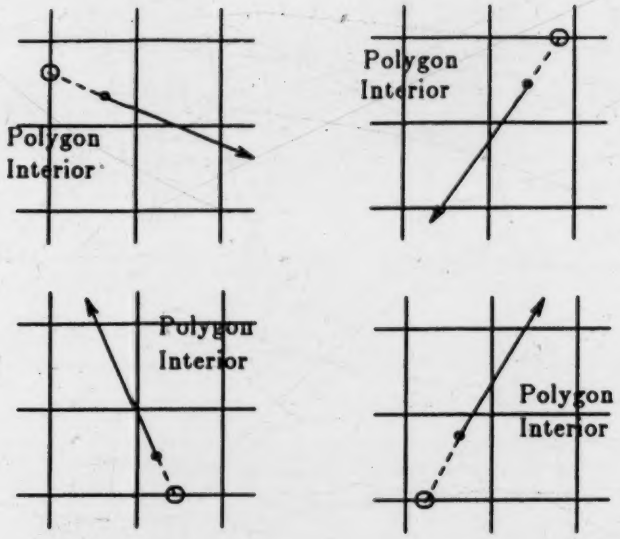


Figure 4.8 Extrapolation of Polygon Edges

The interior of the polygon is labeled in the figure and can be predicted from the assumption that the edges are traversed clockwise. The interpolated point is indicated in the figure with a circle. Notice that the algorithm must determine edge direction before interpolating back to the pixel boundary.

The last function performed by this routine is calculating x_{rem} and y_{rem}. X_{rem} is defined as the x distance remaining between the current point and the next y boundary and y_{rem} is the y distance remaining before the next x boundary. If the interpolated point falls on an x boundary, $y_{rem} = \Delta y'$; if it falls on a y boundary, $x_{rem} = \Delta x'$. Both y_{rem} and x_{rem}, are calculated from the interpolated point in the direction of the edge. The use of these values will be explained below.

4.1.3.2. make_frag

This algorithm is based on a few simple principles. It should be noticed, for instance, that polygon line segments usually have an "axis of greatest change"† ie. Polygon edges will pass through more x pixel boundaries or more y pixel boundaries. This axis of greatest change is reflected in $\Delta x'$ and $\Delta y'$. The greatest absolute value of these indicates whether x is preferred or y is preferred.‡

The other observation, mentioned previously, is that there is a constant x increment between y boundaries and a constant y increment between x boundaries — $\Delta x'$ and $\Delta y'$ respectively.

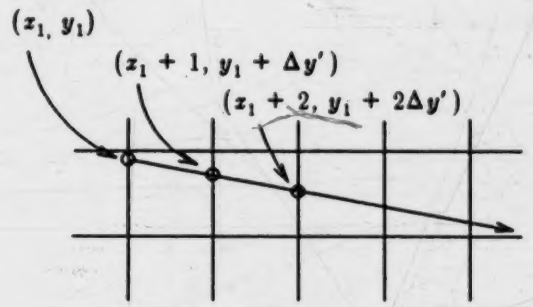


Figure 4.9 Iterative Line Clipping

One can easily calculate the intersections of the polygon edge with the pixel boundaries given the starting coordinates with $\Delta x'$ and $\Delta y'$ — this assumes that a method exists for deciding when the line crosses a y boundary (see Figure 4.9). Since $\Delta x'$ is greater than $\Delta y'$, the x value is incremented by one for each pixel. This is justified since it is known that more x boundaries are traversed than y boundaries and that no y boundaries are crossed in this figure. Therefore the x coordinate calculations are easy. The y coordinates are also easy to calculate because the change in y between x boundaries is simply $\Delta y'$. Incrementing by $\Delta y'$ gives the required y coordinates.

† Unless the angle of the edge is 45°. Special treatment must be given to edges at 45° that also pass through pixel corners.

‡ A direction is said to be preferred if the polygon edge passes through more of its boundaries; thus, when the x direction is preferred, more x boundaries are passed.

To predict intersections between boundaries, x_{rem} and y_{rem} are maintained. These represent the remaining distance between the current point and the next boundary. The values must be updated with each point calculation. So, for instance, incrementing the x coordinate by one means decrementing x_{rem} by one.

These values are used as predictors in the following manner. If the x axis is the major axis of change, then a check is made to see if x_{rem} goes below one. This means that the next y boundary will be reached before the next x boundary. So to calculate the next x value, x_{rem}, the remaining x distance, is added to the current x value. This is done instead of adding one. The y calculation is made easy since it is just the floor of the current y value. The calculation of y_{rem} is somewhat more difficult.

Instead of a simple assignment of $\Delta y'$, as done previously, the y value between the current point and the intersection of the next x coordinate needs to be calculated. In Figure 4.10, this distance is:

$$\Delta y' - \left(y + 2\Delta y' - \lfloor y + \Delta y' \rfloor\right)$$

The whole process is shown in Figure 4.10, in the case of a major x axis change. When the y axis is the major axis of change, the roles of x and y are reversed since the algorithm is symmetric with respect to the two axes.

In the remainder of the algorithm, colors are updated each time a y (or x) pixel boundary is crossed. The same is true of the z value. Bitmap generation, discussed previously, can be done by table lookup or by the algorithm found in [17]. The fragment produced by this phase is 120 bits long (Figure 4.11). It contains two 8-bit x, y values, 24 bits for color, a 16-bit map, a 16-bit Z depth and a 16-bit object ID.

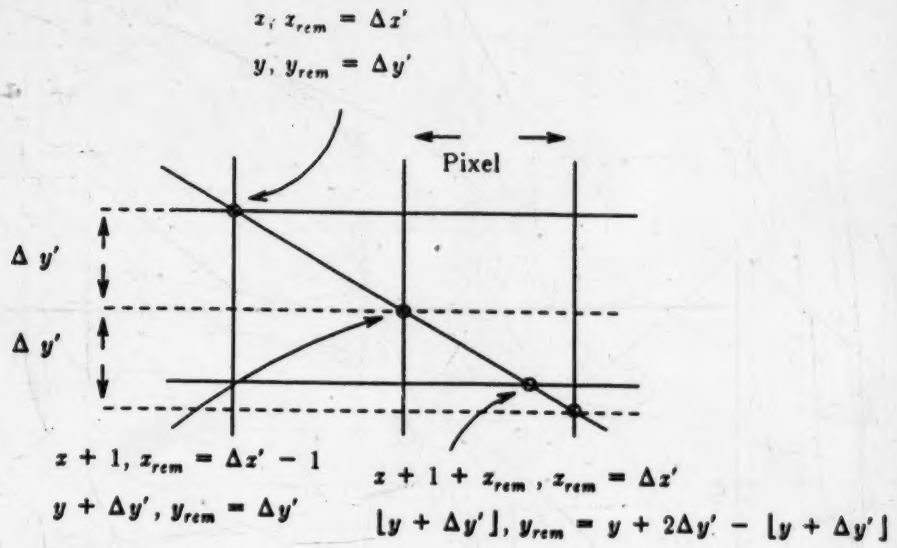

$x, x_{rem} = \Delta x'$
$y, y_{rem} = \Delta y'$
Pixel
$\Delta y'$
$\Delta y'$
$x + 1, x_{rem} = \Delta x' - 1$
$y + \Delta y', y_{rem} = \Delta y'$
$x + 1 + x_{rem}, x_{rem} = \Delta x'$
$\lfloor y + \Delta y' \rfloor, y_{rem} = y + 2\Delta y' - \lfloor y + \Delta y' \rfloor$

Figure 4.10 Clipping Across a Y Boundary

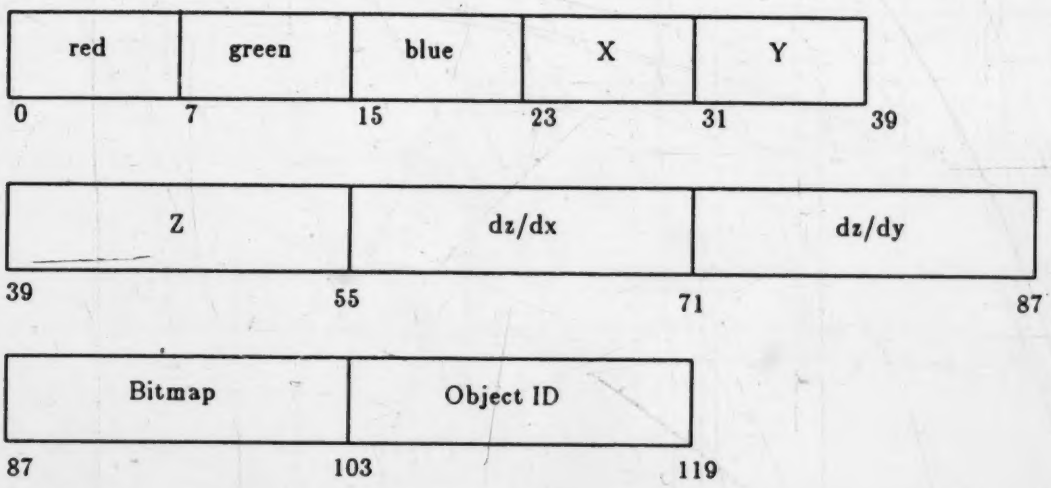


Figure 4.11 Fragment Data Structure

4.1.4. Phase 4

The purpose of Phase 4 is to calculate the color at each pixel on the screen. This is done in several steps. First, fragment lists are sorted in order of depth and common bitmaps are merged. Figure 4.12 is an example of this merging. Here, a polygon corner falls within the boundaries of a pixel (the dashed box) and bitmap generation yields

two fragments corresponding to the corner. A bitwise "AND" operation merges these two fragments into a single bitmap that reflects the geometry of the corner. Note that the crosshatched area indicates the parts of the bitmap set to "1".

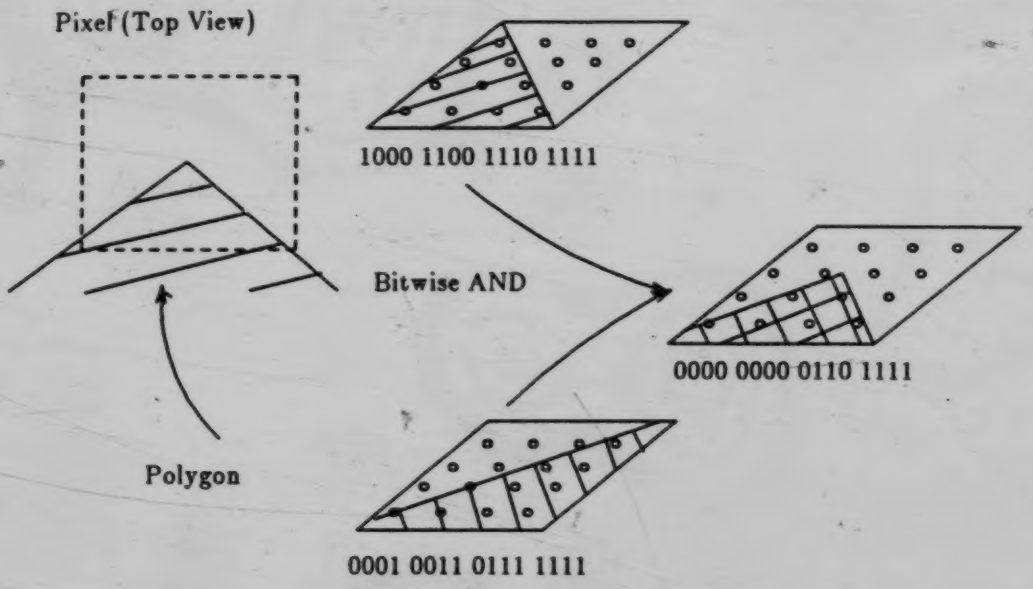


Figure 4.12 Merging Two Common Pixel Fragments

After merging, the subpixels in the polygon interior are retained since these are common to both bitmaps. This procedure also reconstructs parts from those polygons narrower than one pixel width. Bitmaps are known to be common if they possess the same object ID and fall in the same pixel. After the bitmaps are depth sorted and common bitmaps are merged, clipping is done. Figure 4.13 illustrates the clipping phase along with the previous operations of sorting and merging.

Clipping is used to predict how much of a bitmap is visible in a given pixel. Assuming that the top of the figure is closest to the observer, then the closest bitmap "masks out" bits not visible to the observer. The cross-hatched areas of the bitmaps indicate which bits are set. Notice that bitmaps in the dashed box marked "clipping phase" are reduced to only those areas seen from the top of the pixel; thus the background color — completely covered by the previous three bitmaps — has none of its

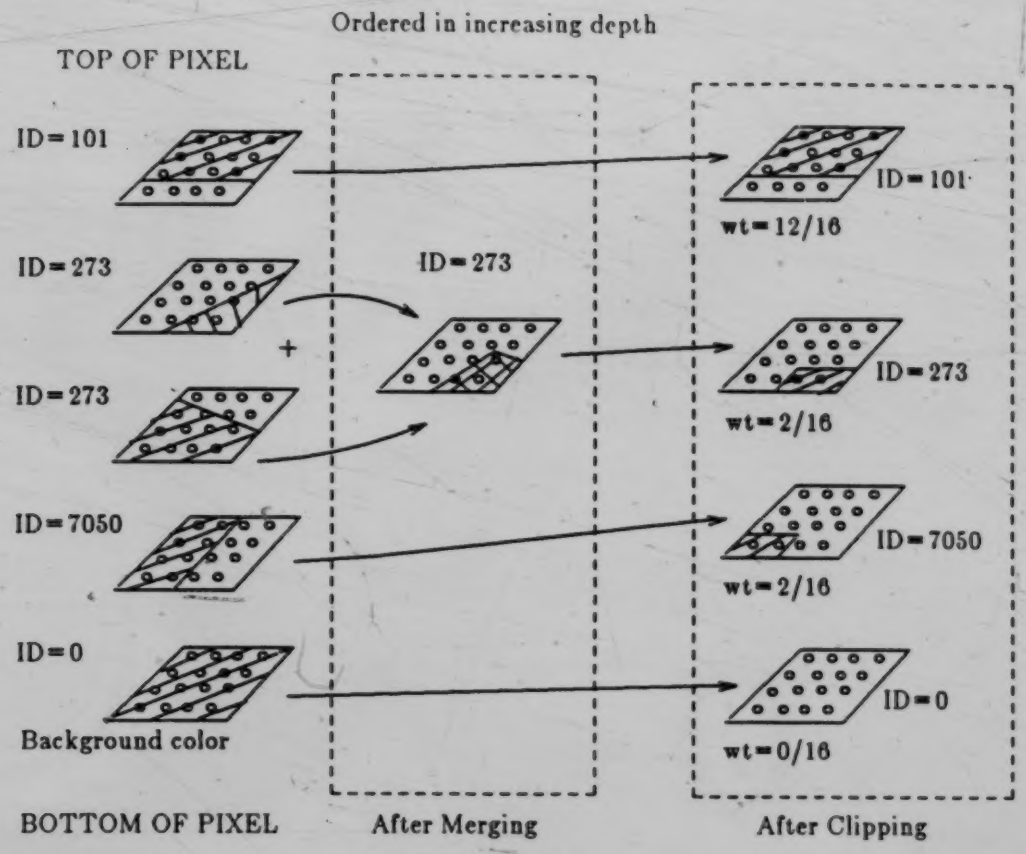


Figure 4.13 Phase 3 Operations

bits set.

After the clipping phase, the bitmaps represent the area and geometry of polygons visible in the pixel. Each bit in these maps represents 1/16th of the total area of the pixel. By summing the number of bits, a weight is constructed for the colors of each bitmap. The final pixel color is determined by summing the weighted bitmap colors. Thus, for object 273 in the figure, a weight of 2/16 is calculated. This means that 2/16 of the bitmap color is contributed to the final pixel.

To continue the example, the red component of the final color is the sum of the partial contributions from all the bitmaps. For the above figure:

$$final\ red = \frac{1}{16} \times \left(12 \times Red_{101} + 2 \times Red_{273} + 2 \times Red_{7050}\right)$$

where the magnitude of red in object x is denoted by Red_x.

This algorithm has a straightforward implementation in a multiprocessor system. By duplicating the code on each processor, it is possible to have processors independently process polygons and store into a common frame buffer. Interaction of fragments for antialiasing is also handled in a parallel fashion since, after data generation, all the data necessary to antialias a single pixel can placed in a "parcel" of work that is easily handled independently. This scheme is elaborated in the next chapter.

The algorithm appears simple enough to implement in VLSI since only fixed point additions are needed. In addition the bitmap generation algorithm has been already been tested. The only complication is the iterative line clipping algorithm which requires some degree of decisional logic.

The above algorithm has been implemented and tested. Images of 256-by-256 pixels have been produced and are found (by inspection) to be comparable to 512-by-512 images produced by the Movie.BYU graphics package.

Chapter 5

The System Architecture

5.1. System Control and Operation

Currently, multiprocessor architectures are being heavily investigated for their commercial potential. Interest in multi-microprocessors is especially great. Although commercial opportunities have not been explored until recently, multiprocessors have been in the literature for many years. Numerous surveys were written on these systems and their interconnection structures [14,15,27,32]. In fact, the field has existed long enough for several textbooks to appear [3,30,42].

The appeal of such systems comes from their potentially great practicality. These systems take advantage of a cheap and reasonably powerful source of processing power. Since system design is usually a modular replication of processors, construction costs are reduced and system repair is cheaper and easier.

A multiprocessor architecture is a good choice for high performance graphics. Since microprocessors are general purpose CPUs, there is little restriction in the tasks they can perform. Special purpose VLSI systems, on the other hand, can only execute hard-wired (and by necessity) simple algorithms. This usually precludes implementing sophisticated algorithms such as anti-aliasing.

By taking a multiprocessor approach to this system design, the problem of insuring sufficient pixel throughput (from a generation standpoint) is eased. If a particular implementation, for instance, has insufficient speed to sustain realtime throughput, processors can be added to augment system capability. This assumes, however, that system design is modular and that significant speedup occurs with expansion. Thus the bandwidth of the particular processing element used is not as important to insuring realtime performance — replication can compensate for a slow processor.

The problem of system design is now reduced to insuring that the memory and busses have sufficient bandwidth to handle realtime pixel throughput. This is an enormous task in itself since potential for "bottlenecking" at the frame buffer end of a graphics system is huge. Consider input data which consists of an average of 4 points per polygon. After pixel production, these points could give rise to a whole frame buffer of pixels (65,536). Thus, data handling requirements for graphics systems are potentially crippling. Careful consideration of these requirements is warranted.

This chapter discusses the system architecture. It provides a high-level view of the system in Sections 5.2 and 5.3. Since the architecture is founded on a distributed-modular arbitration scheme, this is given prominent coverage in section 5.4. Various system components are presented in subsequent sections and the chapter ends with a discussion of system liabilities.

5.2. System Overview

The proposed system is illustrated in Figure 5.1. This system is comprised of four multiport buffers (**AB**) each consisting of a bit-slice processor with memory (**BM**) and four vertical buffer-interfaces (**BI**). These units make-up the frame buffer and are attached to four vertical processing nodes (**PN**) through the buffer-interfaces (**BI**).

Each processing node consists of an arbitrary number of processing boards (**P**) sandwiched between two busses. One of the node busses is connected to the buffer-interfaces of the multiport memory, while the other is connected to a block of high-speed memory (**B**) which buffers polygon descriptions taken from the host computer.

At the bottom of the diagram are some ports connecting a system controller to the multiport buffers. The system controller (**S**) can also write to the host buffers and access status port information. It is responsible for booting the system as well as switching system phases.

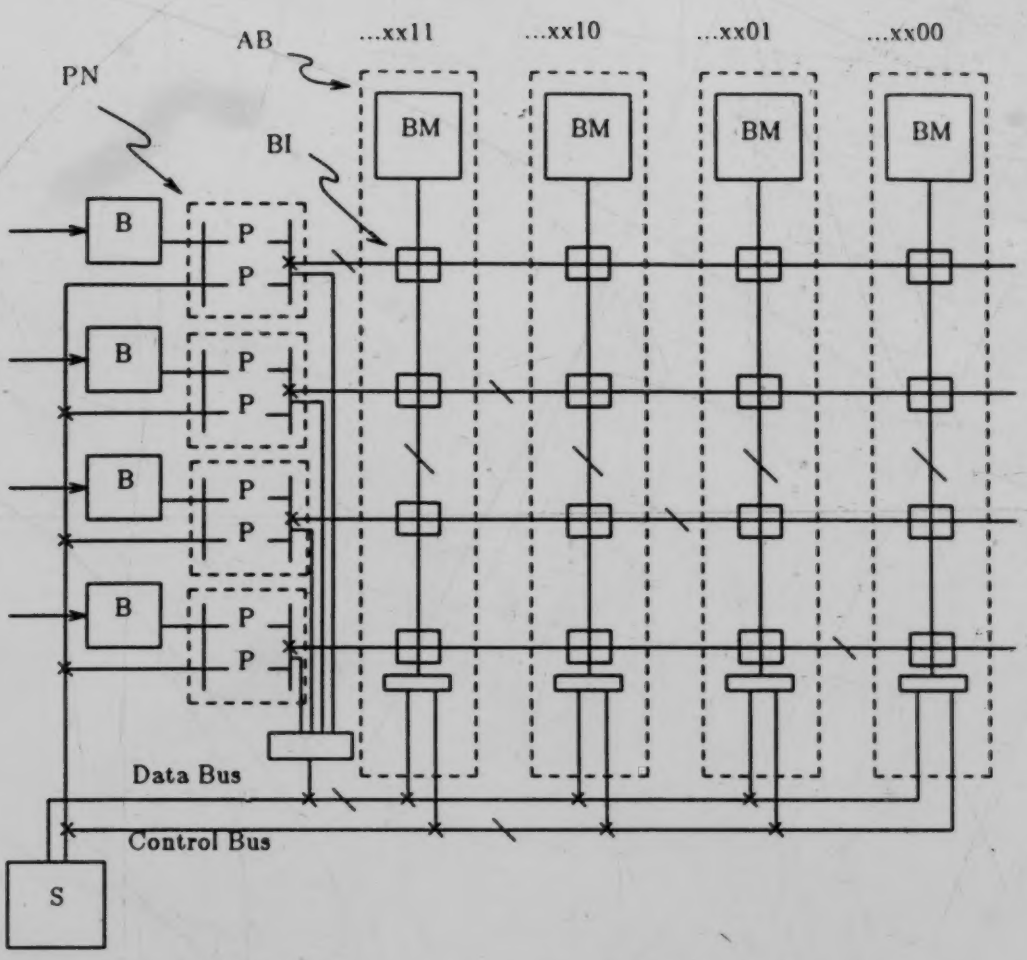


Figure 5.1 System Configuration

There is extensive buffering at each node and at each port on the multiport memory. This allows writing to proceed while the frame buffer is busy and allows the microprocessor to run without blockage.

The system can be "scaled" to allow greater interleaving and addition of more busses to handle greater bandwidth. It is possible to have a "2-by-2" system with two multiport memories connected to two processing nodes. This configuration has a much

lower performance than the illustrated "4-by-4" system, but will be more economical to construct and yet be valuable for a less demanding application. This thesis investigates various scales ranging from a 2-by-2 system to a 16-by-16 system.

The architecture is also modularly expandable since additional processing power can be added at each node by inserting a processing card. The illustrated configuration only shows two processors per node but experiments have been run with as many as 32 processors per node.† The modularity of the system derives from the arbitration scheme explained in Section 5.4.

5.3. System Operation

The following is a high-level description of system operation and uses Figure 5.1. Notice that system operation follows the 4 phase partitioning discussed in Chapter 4.

5.3.1. Phase 1

The system controller (S) starts the bitslice processors (AB) initializing the frame buffer with the background color and depth. At the same time, the host is given permission to start filling the host buffers (B) with polygon primitives. Transfer of polygon information from the host continues into the next phase.

5.3.2. Phase 2

Bitslice processors signal that initialization of the frame buffer is complete. When the last processor indicates finished, the system controller starts the processing nodes (PN). At this stage, polygon descriptions are read from the host buffers (B) into local memory on each processing card. These polygons are initialized for scanline production and the pixel-guns are loaded.‡ Pixel production proceeds with successive writing of

† It is also technically feasible to allow arbitrary expansion of processing nodes with a fixed number of multiport memories. The design of this system allows arbitrary expansion after construction but system "scaling" must be designed and planned before construction.

‡ Pixel guns, as discussed in the previous chapter, are special purpose VLSI devices which produce whole-pixel and fragment data.

whole pixel data to the multiport A-buffer. The phase ends when no more polygons are left to be rendered, the processors are idle and the buffers are cleared.

5.3.3. Phase 3

In this phase, node processors again read polygons from the host buffers. Polygon descriptions are initialized for bitmap production and pixel-guns are loaded. Fragment production results in fragment writes to the multiport memory and the phase finishes according to the criteria given in Phase 2 above.

5.3.4. Phase 4

The system controller signals the start of Phase 4 and node processors begin reading at a "dummy" memory location in the multiport memory. Bitslice processors interpret pixel reading at the dummy location as a signal to send a fragment list. A node processor continues reading at this location until it reads a stop signal; it then releases the bus for another processor.

When a node processor has a complete fragment list, merging and clipping proceeds and the final pixel color is found for that pixel. The completed pixel is written to the multiport memory and the processor attempts to fetch another list.

The system controller terminates this phase when all processors are idle, the buffers are empty and all bitslice processors are signalling finished. A signal is then given to restart Phase 1 and the frame buffer is released to the video display processor.

This is only a sketch of system operation; there are many possible variants using the same basic architecture. Loading of host buffers could commence in Phase 4, for instance. Another possibility is local polygon storage at each processing node. This possibility requires additional memory but eliminates read contention at the "A" bus for Phase 3, and even permits host polygon transfer during this phase. This investigation was designed to examine performance under a basic set of conditions and leaves

"fine tuning" of the system for future work.

5.4. A Pipelined, Modular, Equal-Access Arbitration Scheme

Early in the system design it was recognized that a distributed arbitration scheme was needed to provide system modularity and easy expansion. A search of the literature found an arbiter that provides equal access and modularity but it requires an unacceptable arbitration time [13]. The scheme presented in this paper requires a 40 nanosecond delay to bus access in order to prevent illicit states.

If resource usage is long — in the order of microseconds — then this delay becomes nearly insignificant. But in this system, typical usage lasts 200 nanoseconds. This implies an overhead of 20 percent on each multiport memory access, thus reducing the effective memory bandwidth by 20 percent. Under these conditions the arbiter could not be used for a multiport memory.

It was noticed, however, that arbitration can proceed while the bus is busy. This would allow overlapping arbitration with bus use. The circuit was redesigned to demonstrate the feasibility of this idea. Before proceeding, a problem with "pipelined arbitration" should be mentioned. If the bus is usually idle, then arbitration can not be overlapped with the bus use. Efficient arbiter operation requires the bus to be heavily loaded. The intent of the present design is to flood the busses with pixel write requests. Under these conditions, nearly all the efficiency of the scheme is retained. The following discussion is a description of the arbiter design and its function.

While the claim is made that the arbiter provides equal access, it only does so on a "statistical" basis. At any given time, however, the mechanism selects one device as having the highest priority. This priority rotates among devices to give equality over a long period of time. The "rotating priority token" is accomplished with a "ring counter" described later. The arbitration circuit is shown in Figure 5.2.

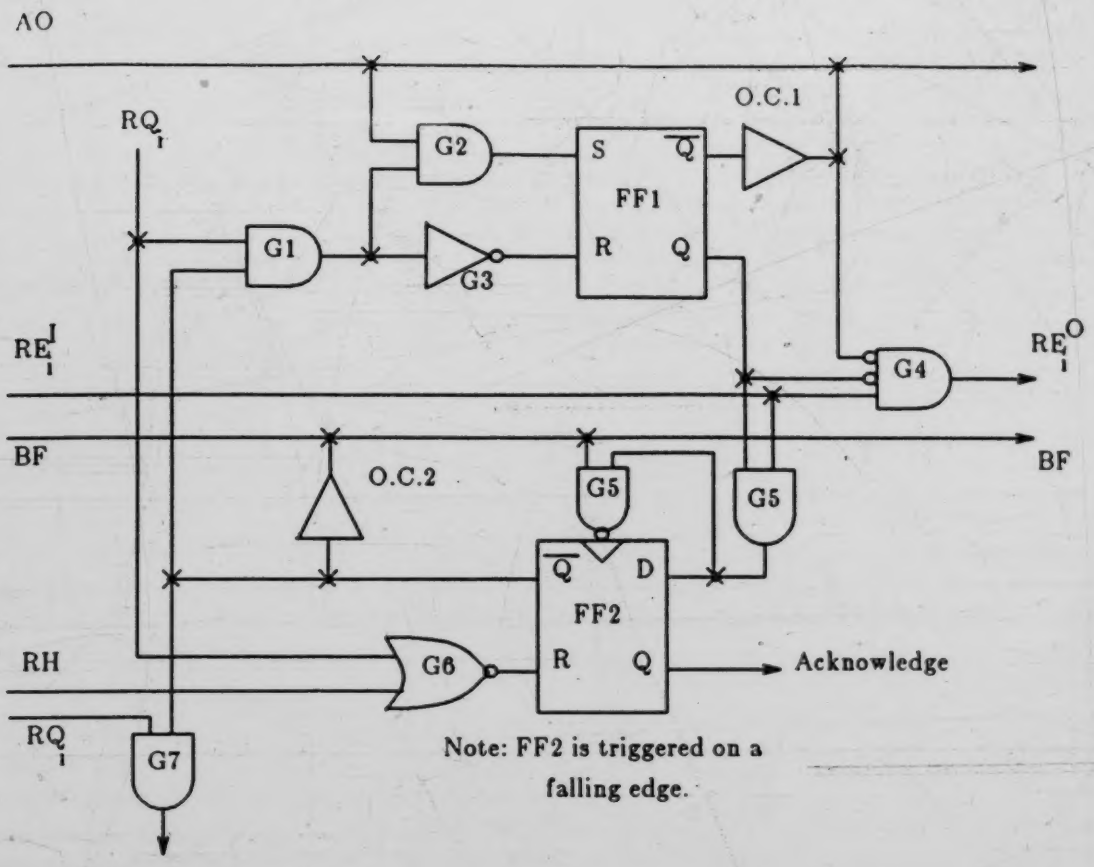


Figure 5.2 Bus Arbitration Circuit

Consider the following sequence of events. While the bus is idle and there are no requests, AO (arbitration open) and BF (bus free) are high while RE (request enable) lines are low. When a request is made at device i, RQ_i (device request signal at the i^{th} device) is asserted high and the wait signal is sent to the requesting device. While this is occurring, FF1 is set and the AO line is dropped by the open collector buffer, OC1. This blocks out any requests that occur after the present one. If other requests are sent concurrent with this one, the other nodes follow the same sequence to this point.

The node currently holding the priority token (PT) asserts RE_i^O high if no request was made at it, since RE_i^O, (the request enable output) passed to the next device, is:

$$RE_i^o = \overline{AO}\ \overline{Q1_i}\,(RE_i^I + PT)$$

and AO is low with $Q1_i$ and PT high. This causes a 1 to be propagated down the chain of nodes until it encounters the first requesting node. Since the Q1 at that node is high, the request enable after that point is held low and nodes after this point are denied access to the bus.

If the bus was previously free then FF2 is set and the request is acknowledged by dropping the "WAIT" line and raising ACK. This also drops the BF signal to indicate the bus is now busy. Another consequence is that the $\overline{Q1_i}$ is set high and the AO line opens arbitration (if there are no other requesting nodes which had simultaneously requested access). Also at this time, the PT (priority token) is shifted in the chain of nodes.

Two things may happen at this point. If AO goes high then there are no other simultaneous requests and arbitration can proceed as described before. If however, there are simultaneous requests, then AO is still low and only the other simultaneous requests are considered for the next arbitration. Since FF1 is reset at the successful node, a high signal is free to propagate past the point of the "current successful" node to the next requesting node. Arbitration is done while the bus is busy. As soon as the bus is set high to indicate "free," FF2 at the next successful node is set and it grabs the bus before opening arbitration for another node.

The circuit described above can replicated and be chained together to almost arbitrary length. As long as the total propagation delay does not exceed 200 nanoseconds, the time taken for arbitration will not significantly be increased (this assumes heavily loaded busses). This design allows circuit boards to be added by simply "plugging them in." No change to the arbitration scheme is necessary since new devices are just inserted into the daisy chain. This allows modular addition of ports on the multiport A-buffer as well as modular addition of microprocessors at the processing

nodes. Figure 5.3 gives a high-level view of this scheme. In the diagram the ring counter is shown separately, although an implementation would integrate it into the arbitration circuits.

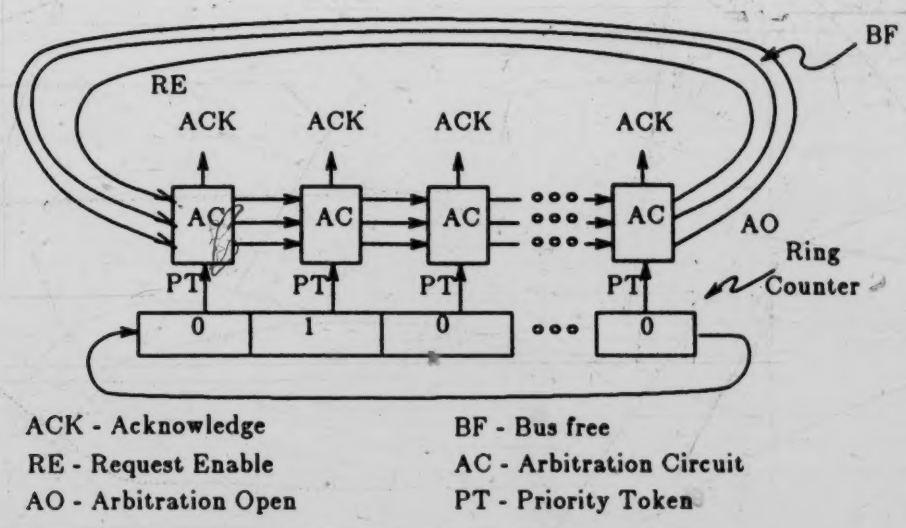


Figure 5.3 Arbitration with the Ring-counter

This arbitration scheme is used in access to all common busses as well as in multiport arbitration. The next section discusses additional functions present in the bus arbiters.

5.5. Interface to Processing Nodes and Requirements

All buffering and bus interfacing in the system can be done with the same component if it's designed with enough flexibility. Specifically, the A-bus buffer, B-bus buffer, and multiport buffers can all use the same arbitration circuitry and the same FIFO queuing structure since all write and read requests to these devices can be handled in the same way.

5.5.1. A-bus Arbitrator

This buffer-arbiter structure interfaces the processing nodes to the memory buffer residing between host and system. This buffer handles 16-bit words and buffers read/write addresses as well. Write operations are buffered similarly throughout the system; namely, when a write operation is performed, the data and address is entered into a FIFO queue by the receiving buffer and an acknowledge signal is sent to the writing processor. When arbitration allows access to common memory, the buffer places the address and data on the address and data lines and performs the write request previously buffered. In writing to memory the buffer-arbiter is transparent to processing nodes since there is no apparent difference between writing to the buffer and writing to memory.

Read requests are handled somewhat differently. To make read requests transparent, the buffer must service requests quickly. It is assumed that a read request has the bus to the buffer tied up. When the buffer gets access to the memory, the read request is sent directly through the buffer to the memory and the request acknowledge is relayed from the memory to the processor. If there is any bus contention before reading, the processor will "think" its dealing with a slow memory device and the interface will remain transparent. Notice, however, that the requesting processor must handle variable read times.

All the buffering logic in the system can be built with the same chip if properly designed. Additional logic is required in the buffer to notify the system controller of any outstanding read or write requests. This is done with a common line which can be set low by an open collector gate at each buffer.

5.5.2. B-bus Arbitrator

This buffer interfaces the processing card to a port on a memory module. It must buffer 16 bit data received from the pixel-gun or microprocessor and send a wider data request to the port. Typically, output from the pixel-gun consists of up to nine 16-bit words. These must be buffered to be sent down the wider data paths of the bus attached to memory. These paths are wider because of bandwidth requirements of the bus, but some multiplexing of data might take place to reduce the path width here as well.

5.6. Processing Nodes

A processing node consists of an arbitrary number of circuit boards attached to two busses — the "A" bus and the "B" bus. Board attachment to the busses is by means of two buffered ports. The buffered ports contain the same arbitration scheme as those in the multiport memory. The processor board also contains a memory mapped VLSI co-processor called a "Pixel Gun."

The pixel-gun operates independent of the CPU so it is capable of storing data to the buffered port while the microprocessor is busy initializing scanlines. There is a memory mapped tristate buffer which isolates the output buffer from the main bus. This is done to permit pixel storage to proceed without disturbing CPU operation.

The system controller has interrupt lines connected to the CPU bus and it communicates specific information (diagnostics, rendering instructions, etc.) by storing the information in the host buffer.

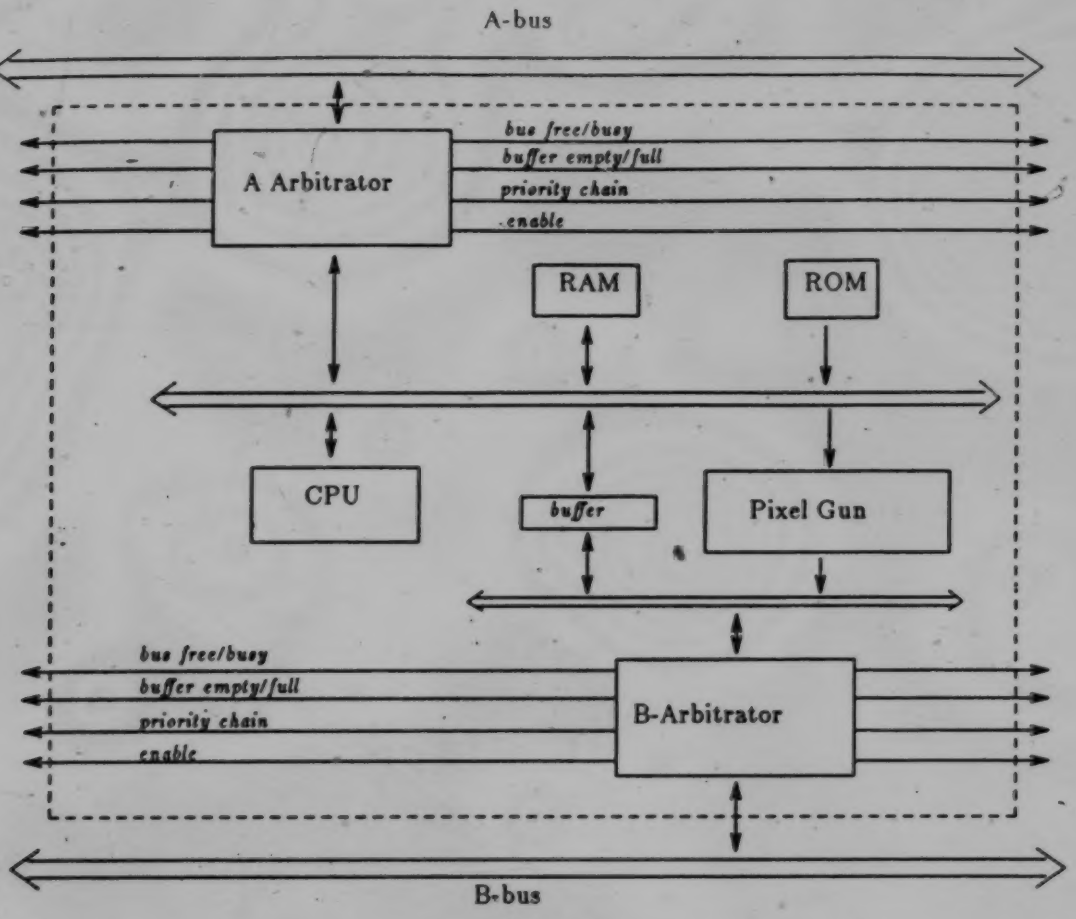


Figure 5.4 Processing Card

5.6.1. The VLSI Processor (Pixel Gun)

The purpose of the pixel-gun is to produce whole pixels and fragments for the system. Since pixels are merely "dots" of color on the screen the most important function of the pixel-gun is to produce the three 8-bit color values of each of the red, green, and blue components. Additionally, the location of the spot must be determined so an x, y address is produced.

The A-buffer algorithm requires more information to calculate the final pixel color. Since it is not known if the produced pixel or fragment is in the foreground or is partially covered (or even totally covered), it is necessary to produce and store the

following data: Z depth values, dz_dx and dz_dy values, the bitmap describing coverage of the addressed pixels (fragment phase only), and an object ID for merging bitmaps from common polygons (fragment phase).

Fortunately, nearly all this data can be produced by integer or fixed point addition. Bitmap production is more sophisticated but the VLSI algorithm is well proven. The fragment generation phase also requires some complexity in decisional logic but these are primarily bit shifting, masking and comparison operations — so all this can be done quite feasibly by a single VLSI device. A more thorough discussion of the algorithm is given in Chapter 4.

The pixel-gun is located at a processing node and shares a circuit board with a microprocessor. It must interface with the microprocessor and be capable of independent data storage into the multiport-A-buffer. CMOS technology is assumed with an estimate of 200 nanoseconds for communication. It is expected, therefore, that the chip can produce partial pixel results every 200 nanoseconds to fully utilize the estimated bandwidth.

5.6.1.1. The Microprocessor Interface

Table 5.1 lists the Pixel Gun pins used to interface to the microprocessor. Not all the pins may be required. If data entry is restricted to a standard sequence then register addressing can be carried out transparently with an on-chip counter; thus eliminating the register select pins. Similarly, the 16 data lines could be reduced to 8 but

Number of Pins	Function
16	Data lines
2	Handshake lines
1	Start processing signal
1	Finished (status from the pixel-gun)
5	Register select (addresses internal registers)
1	Chip select / enable
1	Mode select (Phase 2 / Phase 3)

Table 5.1 Microprocessor Interface Pins

multiplexing requires more data loading operations. Sixteen pins are chosen since they are a good match for the data paths of a 16-bit microprocessor.

Data loading requirements between Phase 2 and Phase 3 vary greatly. The data loaded in Phase 2 is given in Table 5.2. At 200 nanoseconds per load, it takes 1.8 microseconds to initialize the pixel gun for scanline production. Additional data must be stored to

Data	Amount	No. of Bits	Data Type
Color increments	3	16	signed fixed point
Current colors	3	16	signed fixed point
Start_x, Stop_x	2	8	unsigned integer
Current Z	1	16	unsigned integer
Delta Z	1	16	signed integer

Table 5.2 Whole Pixel Phase Loading Requirements

complete a whole pixel but these need not be passed to the pixel-gun as they require no processing. Instead, these values are buffered at the B-bus buffer-arbiter and are copied every time the pixel-gun outputs a complete pixel. These are the current y and the dz_dy, dz_dx values, which are invariant for the duration of the current scanline. This represents an additional 40 bits of information which is loaded after initializing the polygon values (dz values) or loaded prior to each scanline (the current y value).

By contrast the fragment phase has a very large loading overhead. The Phase 3 values that are loaded are given in Table 5.3. This represents 15 loading operations — or 3.0 microseconds to load the pixel-gun for each edge. As in Phase 2, there is data

which remains constant and can be preloaded into the B-bus buffer-arbiter to save time. This data is 48 bits long and represents the ID of the parent polygon and the dz values.

Data	Amount	No. of Bits	Data Type
Color increments	3	16	signed fixed point
Current colors	3	16	signed fixed point
Current Z	1	16	integer
Yrem, Xrem	2	16	signed fixed point
Dz_dx, dz_dy	2	16	signed fixed point
Current x, y	2	16	fixed point
Delta_x', delta_y'	2	16	signed fixed point

Table 5.3 Fragment Phase Loading Requirements

5.6.1.2. The Buffer-Arbiter Interface

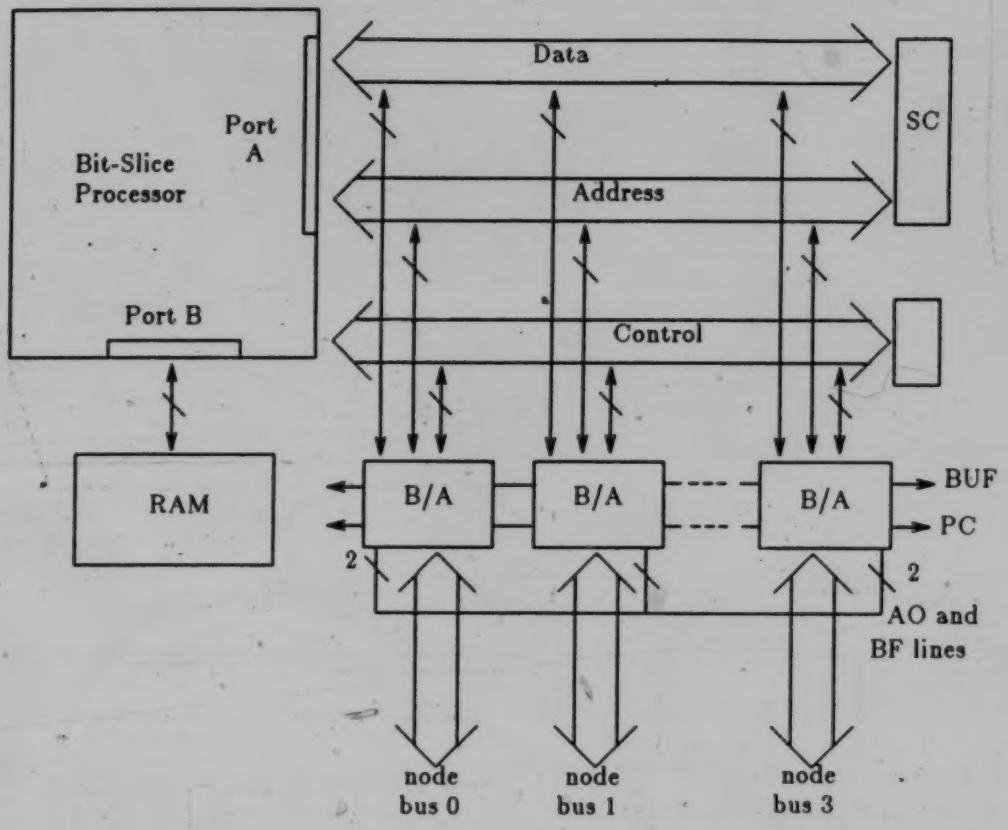
As stated previously, the pixel-gun must be capable of independently storing pixels and fragment information. The interface assumes a 16-bit data path, two bits for handshaking, a three-bit address, and a one-bit signal to tell the buffer that the data is complete and should be sent. Table 5.4 lists the pins used to interface the B buffer-arbiter to the pixel-gun.

Number of Pins	Function
16	Data lines
2	Handshake lines
3	Address select
1	Send data

Table 5.4 Output Buffer Interface Pins

5.6.2. A-buffer Structure and Function

Figure 5.5 is a schematic of the A-buffer processor and memory. The "A-buffer" consists of a high speed bit-slice processor which interfaces to the memory on one side, and to the buffer-arbiters on the other. Processor and arbiter are transparent to the processing nodes which perform read/write operations on the memory.



SC - System Controller
MBE - Master Buffer enable
BUF - Buffer Full Signal
PC - Priority Chain

Figure 5.5 A-Multiport Buffer

The processor responds differently to the incoming data depending on what phase is currently being executed.

The primary purpose of this processor is to compare incoming data to resident data and either accept or reject it on the basis of depth. This processor also retrieves linked lists of fragments from the memory for merging and is responsible for maintaining and linking incoming fragment data.

The memory acts as a frame buffer and an unstructured storage area for the fragment data. It is therefore heterogeneous in structure since it must interface to the CRT with the simple color data and yet store more complicated data types such as fragment structures. To achieve high-speed performance, the frame buffer portion of this memory must be "dual-buffered" to allow the display processor to refresh the CRT without affecting pixel storage. This system assumes a memory speed of 200 nanoseconds. A description of A-buffer operation follows.

5.6.2.1. Phase 1

In phase 1 the bitslice processor gets the background color of the frame and initializes all the module pixels to the appropriate color, depth and dz values.

5.6.2.2. Phase 2

In this phase whole data types are stored into memory. The bitslice processor reads data on the A-port and fetches the current pixel depth corresponding to the incoming data. It then does a comparison of depth values and either rejects the arriving data or stores it over the current data. No reading is done by the node processors during this phase. It is assumed that comparison of Z values is very fast, since ECL logic is capable of comparison in under 12 nanoseconds (Motorola ECL Data Guide).

5.6.2.3. Phase 3

Fragment data types (bitmaps) are created and stored during this phase. The storage operation is analogous to the above but depth comparison is more complicated since a bitmap does not cover the whole pixel. Depth comparison consists of using dz values to find the closest extent of the fragment and the furthest extent of the whole pixel data. A comparison of these depths leads to either trivial rejection of the fragment bitmap (it is covered) or conditional acceptance — the bitmap covers the whole pixel data but may be covered by other fragments.

Acceptance is more complicated since the fragment structure must be stored into two linked lists for easy retrieval in Phase 4. The times taken for these operations are coarse estimates since there is no working model and no easy way to benchmarks the proposed processor short of building, or at least designing it. Assuming ECL high-speed logic, a comparison time of 650 nanoseconds and an insertion time of 2 microseconds is assigned. These hopefully represent realistic (and possibly pessimistic) estimates for performance.

5.6.2.4. Phase 4

During this phase, linked lists of fragments and whole data are retrieved from memory. Since reading processors have no information on where the aliased pixels are, the bitslice processor must maintain and keep track of them. This retrieval is transparent to reading processors since they all read a specific memory location until incoming data indicates a stop. Because the memory processor "knows" the current phase, reads are interpreted as requests to send fragment lists. Fragment lists are fetched by starting at the next unmerged-aliased pixel and sending its fragment list to the currently requesting processor. The current phase is completed when the last-aliased pixel list is sent. Requesting processors are then sent a "finished" signal. When all processors have received this signal and all buffers are empty, the system controller

starts a new frame.

5.6.3. Memory Access Patterns of the Algorithm

Some account has to be made of how the memory is to be configured since one would like to avoid unequal usage of the interleaved frame buffers. The question is how to partition, or interleave the memory modules in order to insure an equal distribution of work. It was decided to interleave the memory on the lower address bits as opposed to the highest order bits.

This decision solved two problems. The first was the non-random nature of scene data. Visual complexity of scenes tends to "cluster," so certain areas of a screen image can contain thousands of polygons, while others may contain only one or two. If partitioning is on the highest order bits, it is possible that one memory module will have nearly all the activity and storage while the others sit idle. Thus, interleaving on the lowest order bits insures uniform image dispersion throughout the memory modules.

The other potential problem is "temporal" clustering of data. This occurs if pixels are consistently sent to the same memory module in roughly the same period of time. Consider a worst case situation in a simple system interleaved on the last bit. If all even addressed pixels are produced followed by all odd pixels, one node will be alternately hit while the other remains idle. Fortunately, "scanline order" production causes memory modules to be addressed in a consecutive fashion as pixels are produced — only in rare cases will a single module be consecutively accessed by a particular working processor.† By interleaving on the last 4 bits, work is evenly distributed in time as well as space — this observation gives great promise for system efficiency.

---

† In phase 3, some temporal clustering can occur. Vertical edges will fall into one module and could cause workload distribution problems.

5.7. System Limitations

There are numerous limitations in the system described. Some of these are limitations in the algorithm, while others are implementation decisions.

The current version of the system does not support transparent objects although these can be accommodated in principle by the algorithm. A more sophisticated implementation of this algorithm should permit this.

There is an upper limit on the number of fragments the system can handle since each module has only a finite amount of memory. This, indirectly, places an upper bound on the number of polygons that can be handled.

The design of the system rests heavily on its ability to store many intermediate results in the form of whole pixel data structures and bitmaps. This requires a large amount of RAM and can make implementation costly. It is hoped that the ever decreasing cost of memory and VLSI will make the economics of implementation much more feasible.

Chapter 6

Simulation Results and Discussion

The primary question to be answered in any architecture study is: "How well does it perform"? Despite the simplicity of the question, it subsumes many other questions. How well will the architecture perform with various data? What are the bottlenecks in the system? Are the bandwidths of the various components well matched? How well is the work distributed in the system? This thesis set out to determine the answers to these questions.

A very important issue in this study is the expandability of the architecture; specifically the concern is getting linear speed-up. This involved investigation of performance gains by increasing the number of processors in the system and by scaling the system to various configurations. Important obstacles are identified in achieving speed-up potential and these issues are considered first in this chapter.

The tools and methodology used in the experiments are discussed in the appendices. This includes a description of timing parameters, data characteristics and level of simulation. The interested reader is, therefore, directed to the appendices for this material. In most cases no knowledge of simulation detail is needed to appreciate the results, but some results require consideration of the methodology and this is provided when necessary.

6.1. Speedup with Added Processors

To investigate system performance with added processors, test sets were created with high aliasing and moderate aliasing. These were used with two natural scenes to observe speed-up in a 4-by-4 architecture.† In Figure 6.1 the time to produce one frame is plotted against the number of processors per node. There is no great difference

† Some of the characteristics of these data sets are given in the appendix.

between randomly generated data and data derived from a natural scene. The plant scene follows Test Set #1 closely This lends credibility to results gained from a random source.

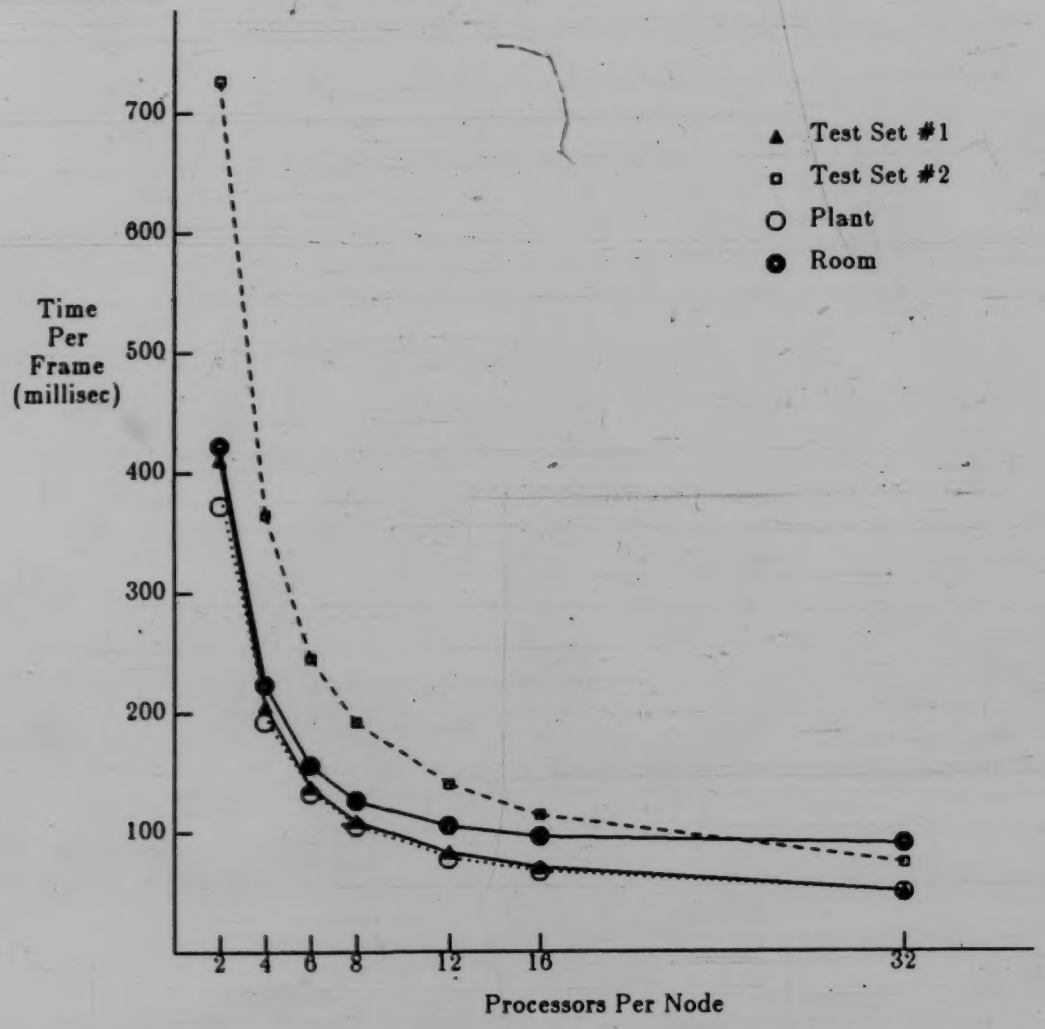


Figure 6.1 Processors Per Node vs. Time

Test Set #2 made the most dramatic performance gains (Figure 6.2). Unfortunately, speedup for all data sets is below the theoretical maximum. There is clearly a departure from linearity past six processors per node. Again, Test Set #1 behaves like

the plant scene. Performance was investigated further by plotting speedup on a phase basis. There is a wide diversity in performance gained using the four data sets. The room scene has particularly poor speedup. Reasons for this are considered later in the chapter.

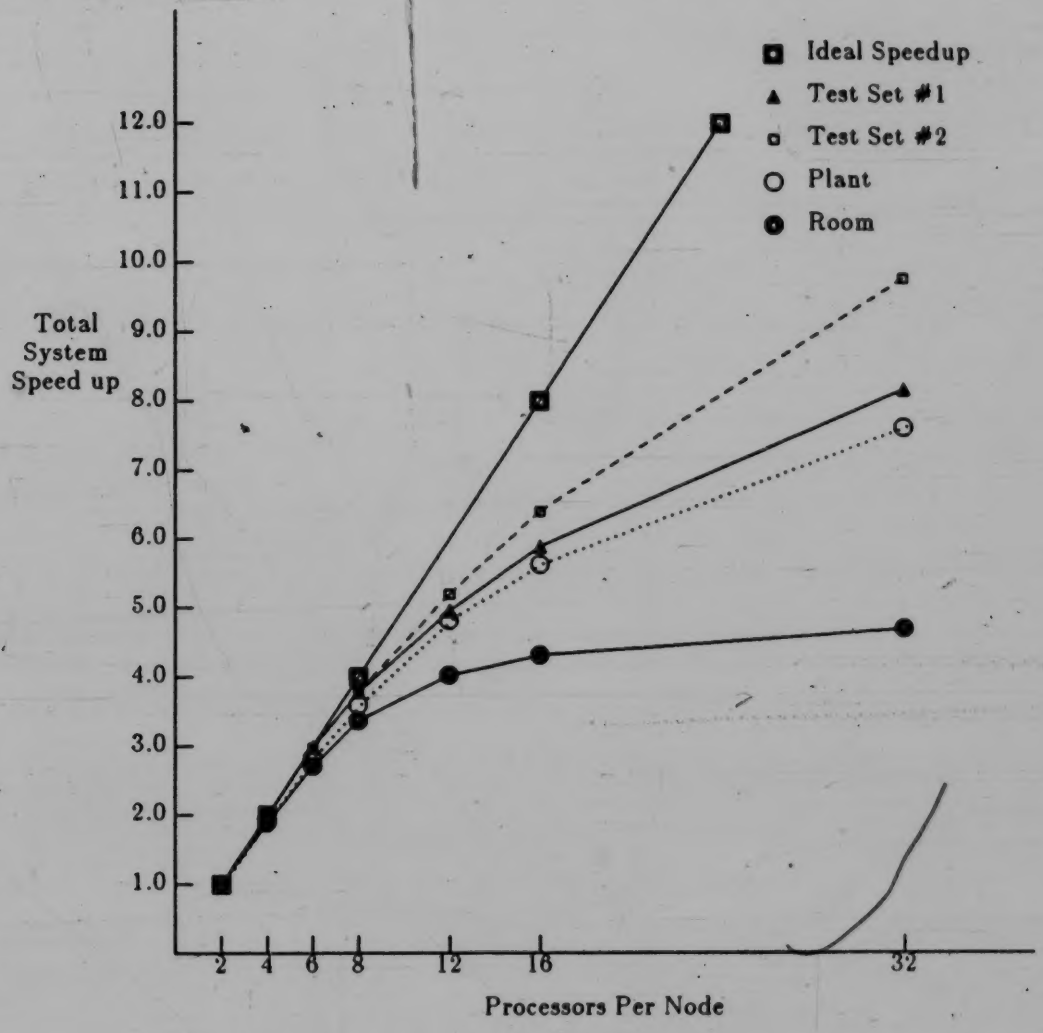


Figure 6.2 System Speedup vs. # of Processors

Since there is little difference between the data sets (as far as the relative performance of each phase) only Test Set #1 was plotted. This graph is given in Figure 6.3.

Phase 2 (whole pixel production) and Phase 4 (fragment list merging) give almost perfect speed-up. The performance of Phase 4 is almost perfectly linear.

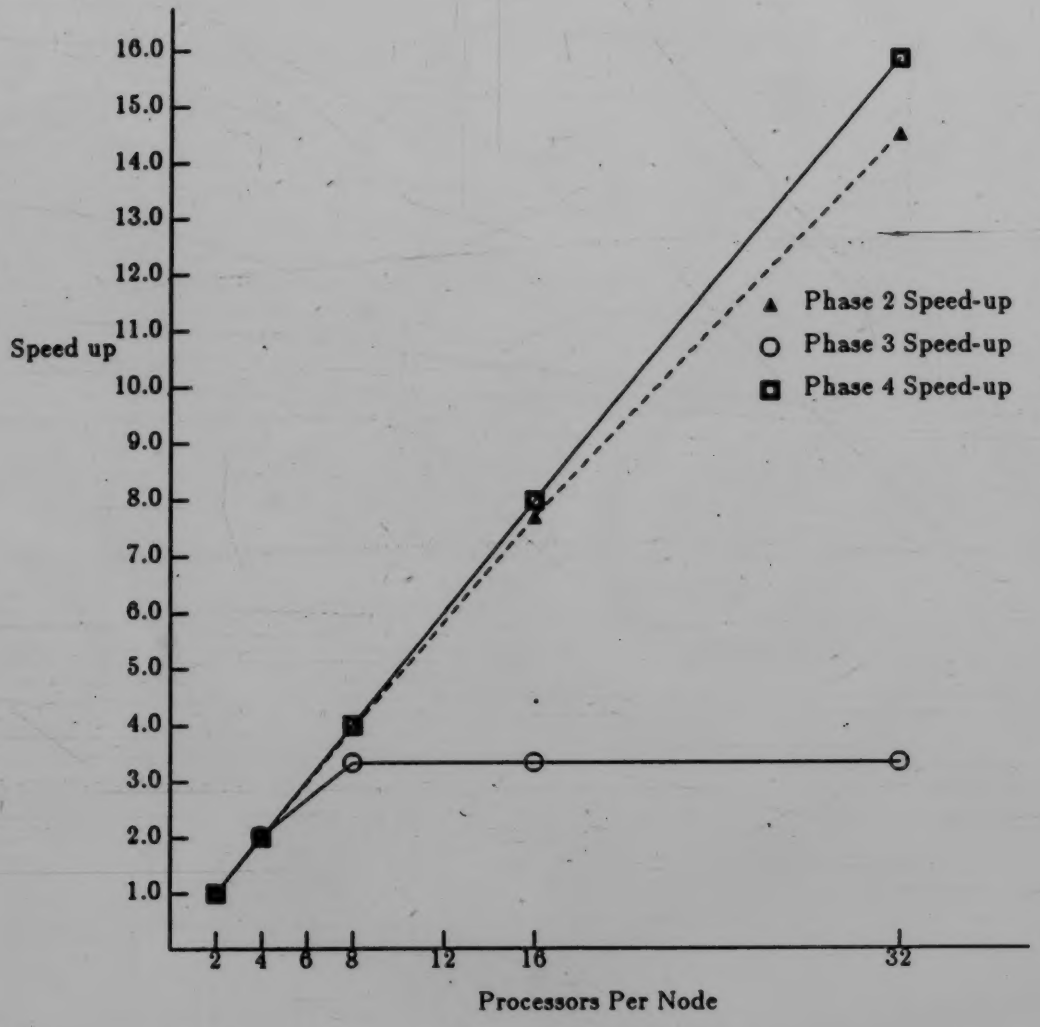


Figure 6.3 Speedup vs. Processors For Test Set #1

Phase 3 apparently meets a bottleneck since there is absolutely no performance increase past 8 processors per node. To determine why these phases performed as they do, the main performance indices were plotted against increasing processors for each phase.

Figure 6.4 gives percentage utilization for CPU, memory, pixel-gun, A-bus and B-bus. The first observation is the remarkably low use of all the system components except for the CPU, which operates at maximum capacity. This is strong evidence for microprocessor speed being a bottleneck in Phase 2.

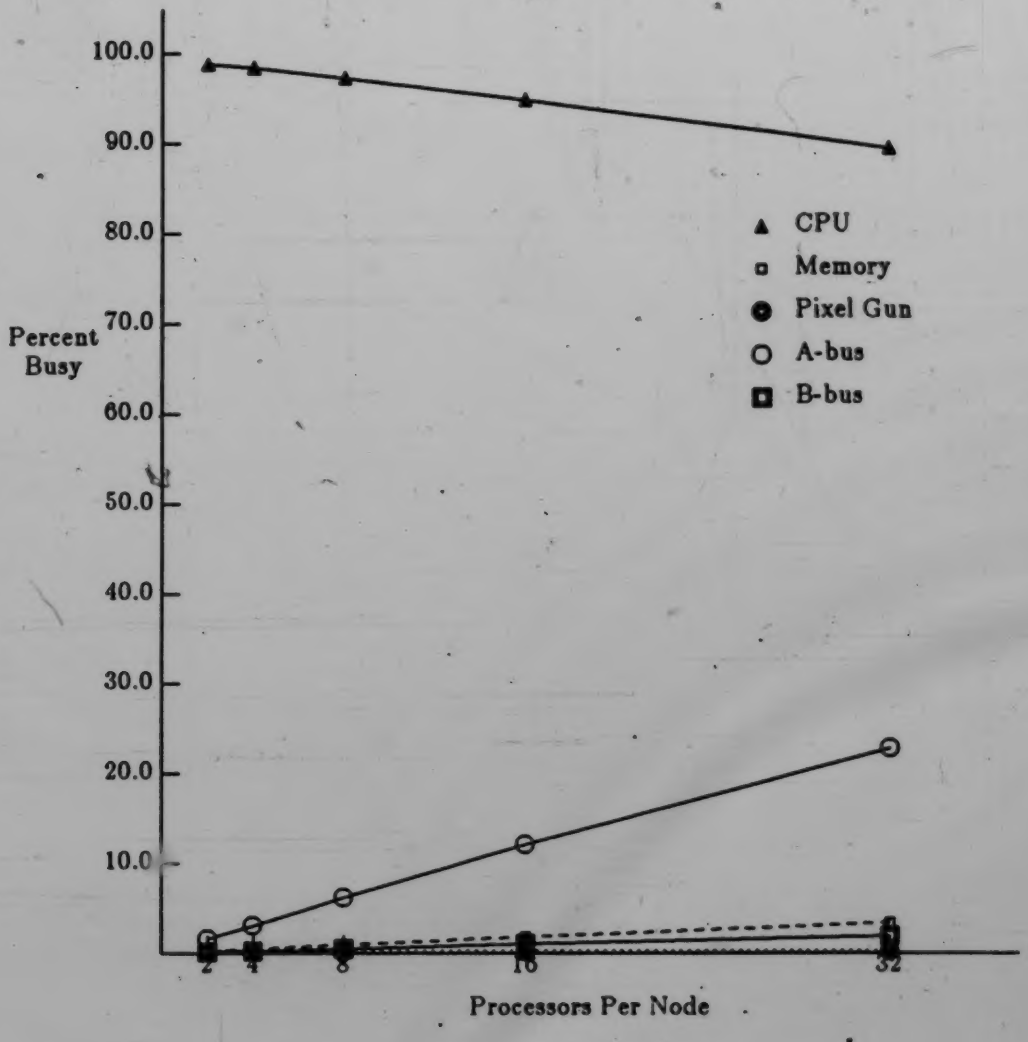


Figure 6.4 Phase 2 Performance vs. Processors For Test Set #1

This graph also shows almost perfect linearity of busy time with added processors. The memory and bus use show that the number of generated and stored pixels,

handled per unit time, is doubled by doubling the number of processors. Table 6.1 better illustrates this linearity.

A slight degradation of CPU is noted with an increase of A-bus use. This suggests there might be some contention in reading polygons from the host buffer. Loss of CPU use may not be as bad as suggested, since statistics are collected from the beginning of Phase 2 and includes the idle time of all processors waiting for their first polygon. This idle time appears to increase in proportion to the number of processors in the system. If more polygons were rendered, this initial idle phase would diminish in comparison. Results collected during Phase 4 support the idea that this is a simulation artifact rather than a problem with contention.

Parameter	Number of Processors				
	2	4	8	16	32
CPU	98.8	98.4	97.3	95.0	89.7
Memory	0.235	0.468	0.933	1.800	3.410
Pixel Gun	0.155	0.154	0.154	0.146	0.137
A-bus	1.57	3.12	6.18	12.1	22.8
B-bus	0.127	0.262	0.492	1.02	1.93

Table 6.1 Percent Busy of Performance Indices in Phase 2

Results are plotted for Phase 3 and are given in Figure 6.5. Fewer than 4 to 6 processors per node give nearly linear speed-up as indicated by the increase in memory use. Between 6 to 8 processors however, memory reaches its maximum capacity. At this point the B-bus is also saturated since write requests will hold the bus until the receiving buffers can hold the request.

The initial busy time of the memory starts at 30 percent while the B-bus uses only about 2.5 percent of its capacity. This suggests a major performance mismatch deriving from the assumption that it takes 2 microseconds to store a fragment and 650 nanoseconds for rejection. This is much more than the 200 nanoseconds taken to transfer fragments on the B-bus.

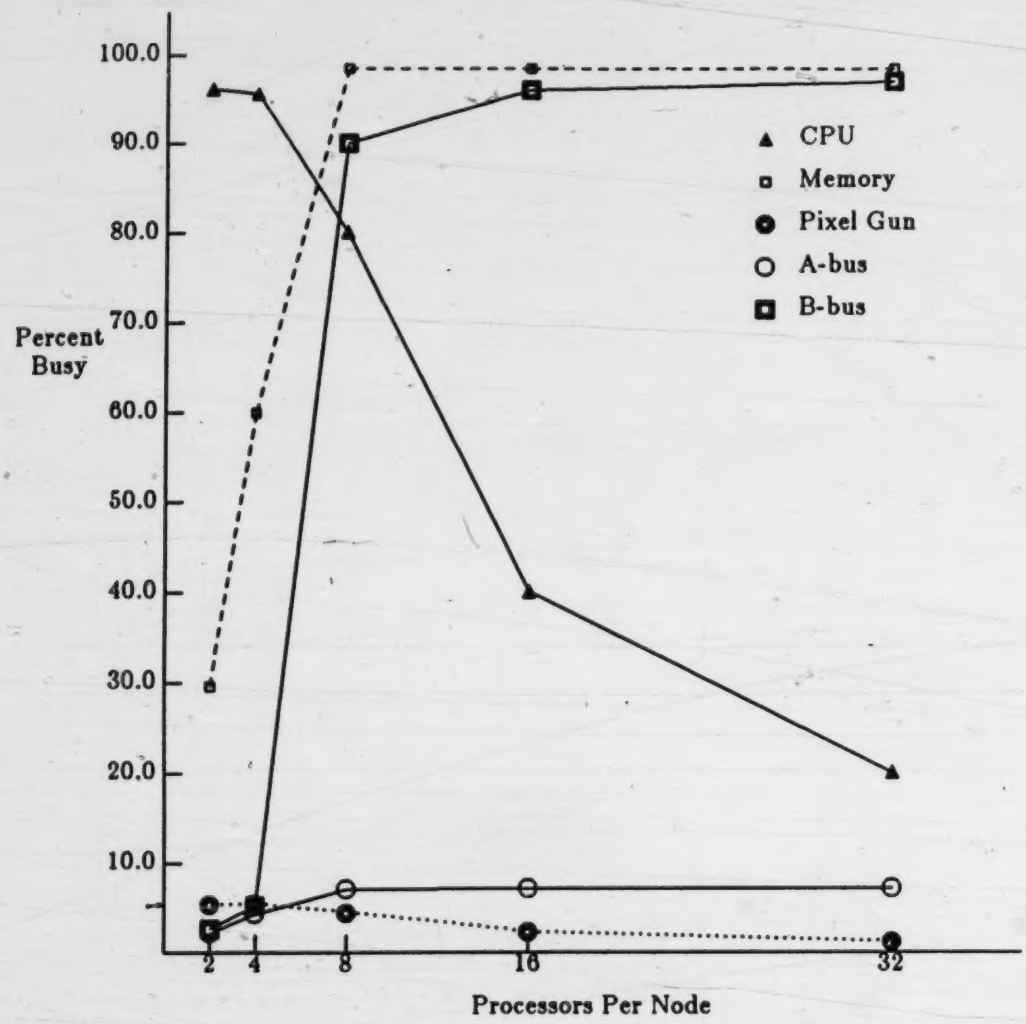


Figure 6.5 Phase 3 Performance vs. Processors For Test Set #1

Another observation is that percent CPU busy does not degrade as rapidly as other parameters. This is partly due to the buffering capacity of the arbiters at each processing node. But the dominant factor is likely to be the CPU production rate which is slow compared to the transfer and storage rate of fragments.

Phase 4 performance characteristics show good speed-up with added processors (Figure 6.6). Nearly perfect increase in B-bus busy and memory busy indicates a nearly

linear increase in fragment list merging. An important observation is that CPU busy is maintained at nearly 100% capacity despite B-bus loading to a maximum of about 52%.

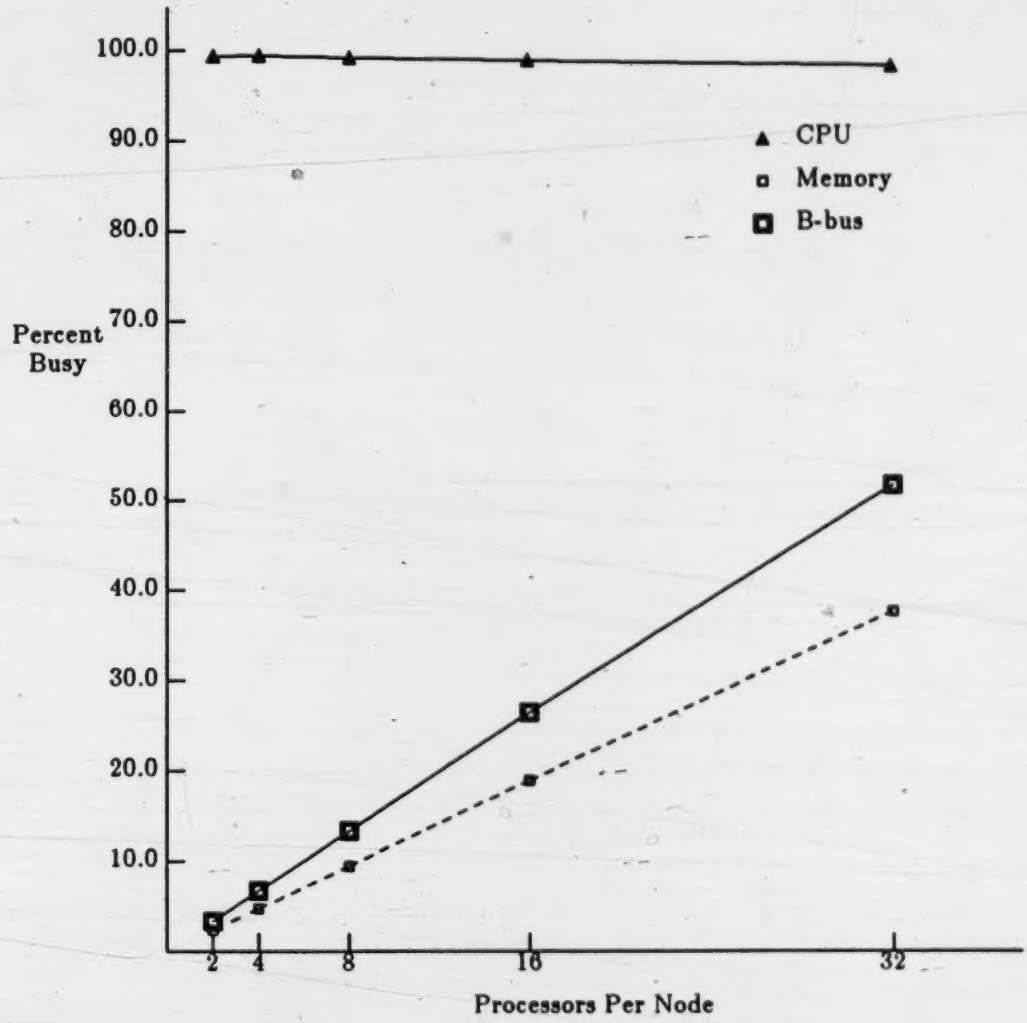


Figure 6.6 Phase 4 Performance vs. Processors For Test Set #1

In fact, CPU busy degrades only about 0.8% from 99.4% (in the two processor case) to 98.6% (in the 32 processor case). This clearly supports conclusions about processor busy in Phase 2. Namely, that CPU degradation is an artifact of the initial reading

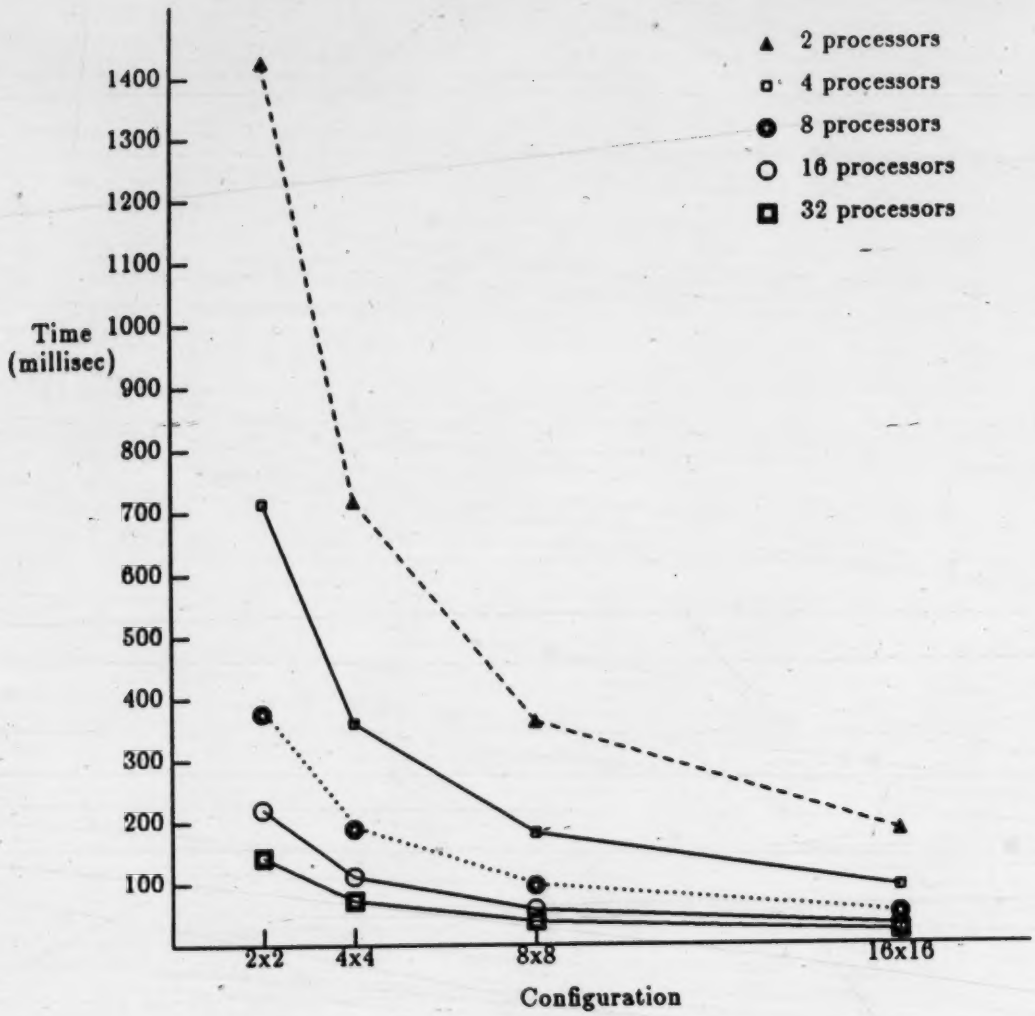


Figure 6.7 Time vs. Configuration for Random Polygons

contrast to the previous experiment, Phase 2 now has minimal performance gains and Phase 3 has the best performance increase of the simulated phases. Table 6.2 gives utilization for Phase 1 system resources. Despite zero performance gain with increasing system configuration, all utilization figures drop. In fact, after the 4-by-4 configuration, utilization suffers a linear decrease in all categories.

In addition, time required to complete Phase 2 is constant for configurations

phase when all processors stand idle waiting to read their first polygon. This is not a problem in the fourth phase since there are tens of thousands of fragments to be read and the initial idle phase is, therefore, much smaller in comparison.

To summarize, these experiments suggest a serious performance mismatch in Phase 3 limiting potential for linear speedup. Specifically, fragment handling capacity of the multiport buffer is exceeded after 6 processors per node.† There is also evidence that CPU capacity is a bottleneck in the system.

6.2. Speedup With System Scaling

In principle, scaling should give perfect speedup since processors, memory and busses are added in direct proportion to each other. Thus, the load balance between system resources should be unaffected. This prediction is confirmed by experiments with random polygons. Time is plotted for 5 different cases against increasing configuration size in Figure 6.7.

Plotted speed-up for these configurations is given in Figure 6.8. There appears to be a slight performance degradation with increasing processors per node, but for the two and four processor per node case, performance gains are very good. The same experiments were done with the room scene and the time plot is given in Figure 6.9. There is apparently little improvement for the 16 to 32 processor per node case. The minimum rendering time appears to be about 60 milliseconds. There also appears to be degradation in speedup when increasing the system scale since all curves appear to converge at the 16-by-16 configuration.

Speedup relationships are better illustrated in Figure 6.10. For 16 to 32 processors per node, speedup barely exceeds a factor of two. Performance was examined on a phase basis using the 16 processor per node case and this is given in Figure 6.11. In

† Six processors per node is not an absolute number independent of processing speed. It is, rather, a function of processor speed.

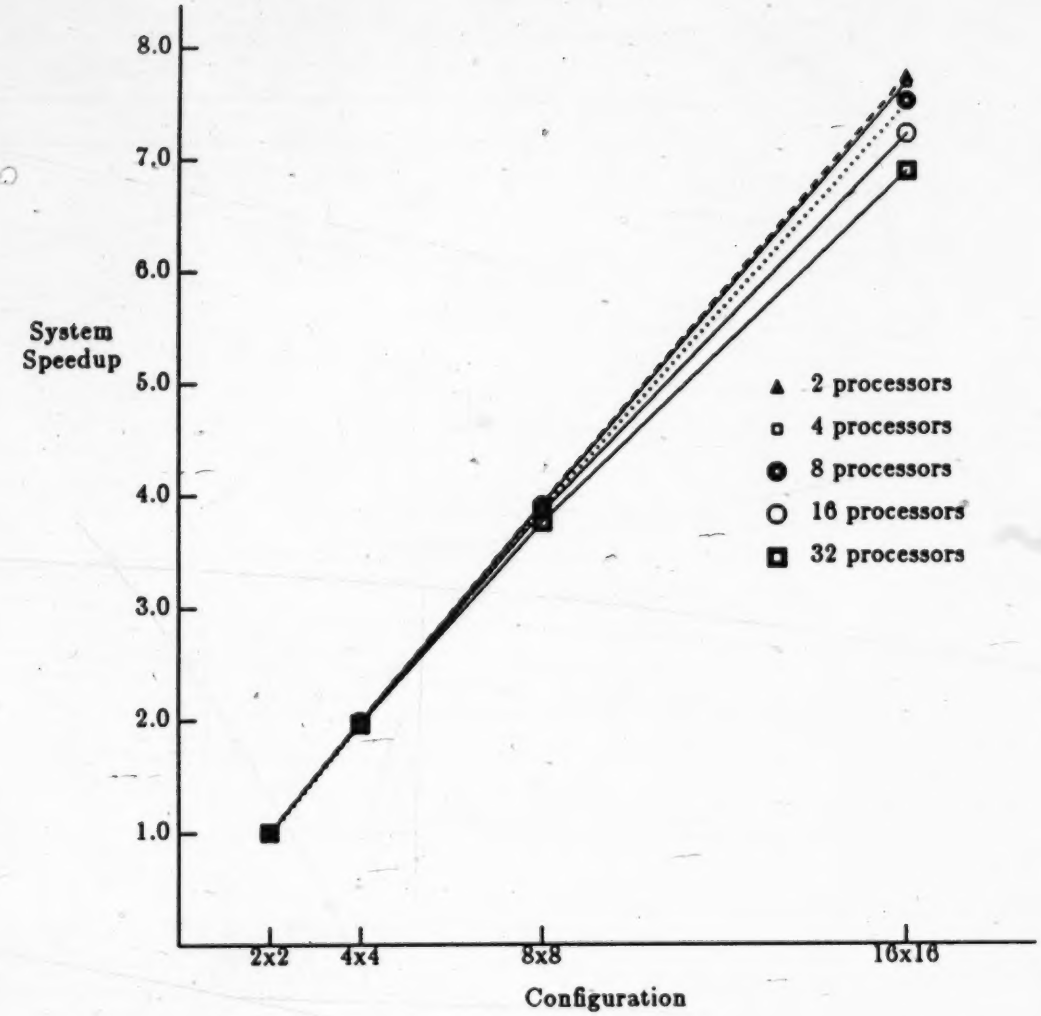


Figure 6.8 System Speedup vs. Configuration For Random Polygons

greater than 4-by-4. Phase 3 only has a slight utilization loss while Phase 4 utilization drops to half for the memory and B-bus.† Lack of speed-up is not due to resource contention problems (as was the case in previous experiments).

If no single resource is causing the bottleneck, it is suspected that a single device or devices could be the problem. This will occur if there is a non-uniform workload

† Memory utilization is 14% in the 2-by-2 case, and 7.5% in the 16-by-16 configuration.

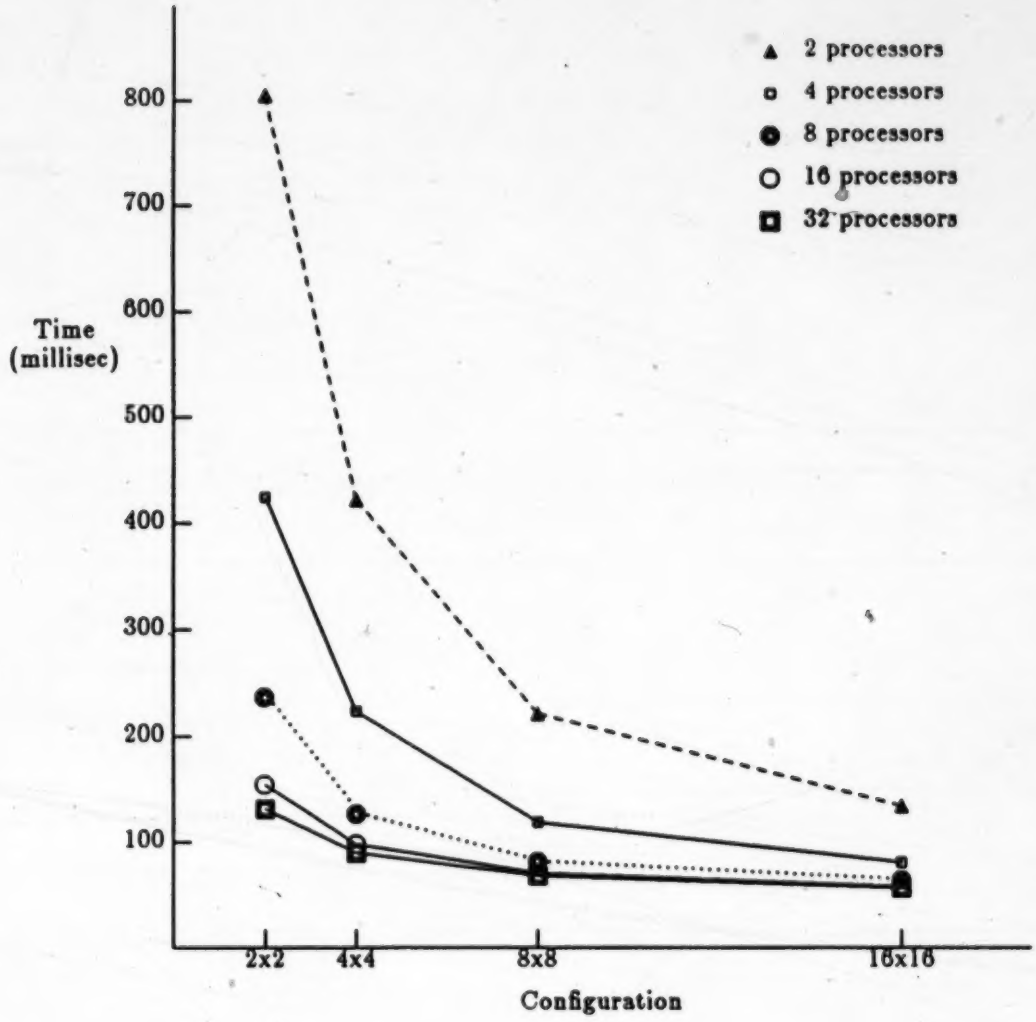


Figure 6.9 Time vs. Configuration for The Room Scene

distribution in the system and single processors or other devices are handling more work than the others. Thus, as the system is scaled upward, added devices become increasingly idle.

Polygons in the room scene are very heterogeneous, ranging from tiny polygons which makeup a vase, to huge polygons that define the floor and walls of the room. It is therefore hypothesized that one or two huge polygons were limiting the system

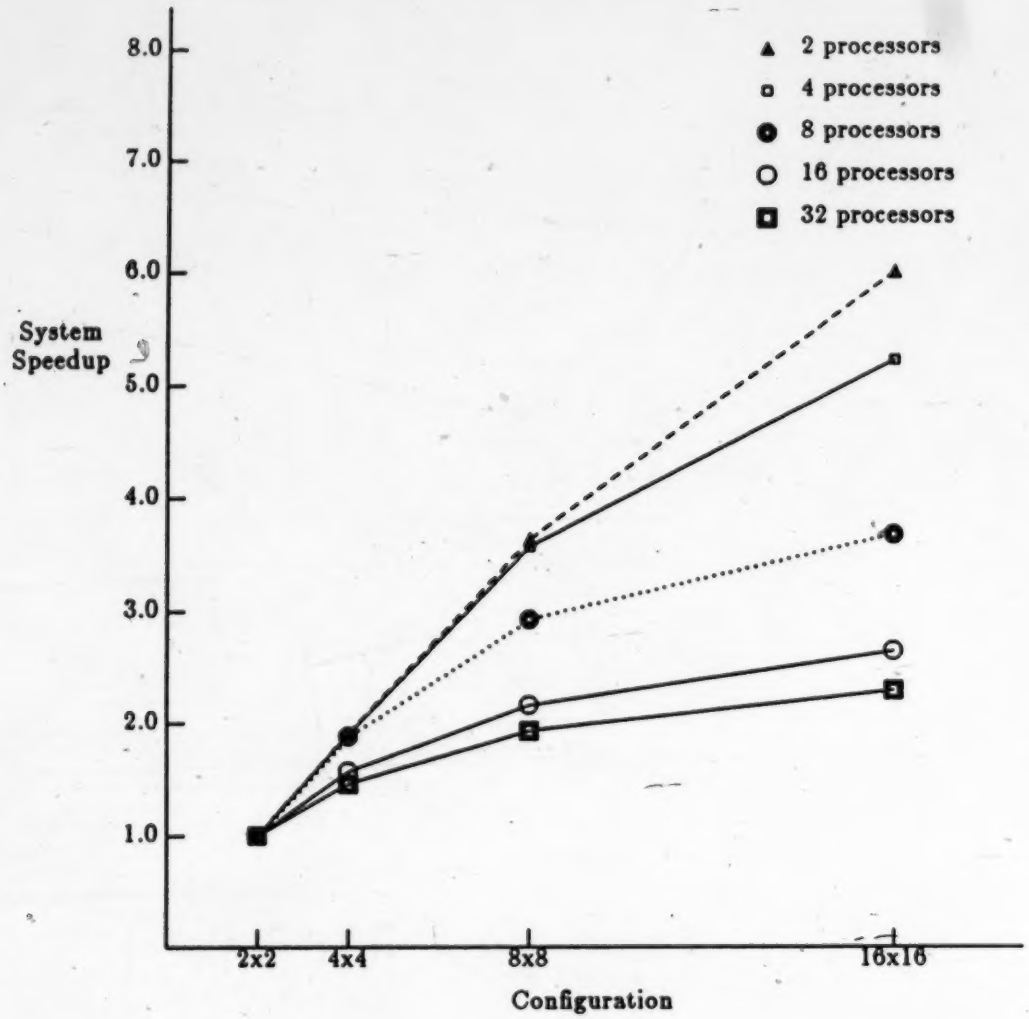


Figure 6.10 System Speedup vs. Configuration For the Room Scene

speedup This conjecture was based on the observation that a polygon represents an indivisible unit of work in the system. No speedup could be obtained by adding processors if this "unit of work" dominates completion time of a phase.

This may explain why Phase 2 time is constant past the 4-by-4 configuration. One processor may be initially loaded with a huge polygon which it works on well after all the other processors have finished.

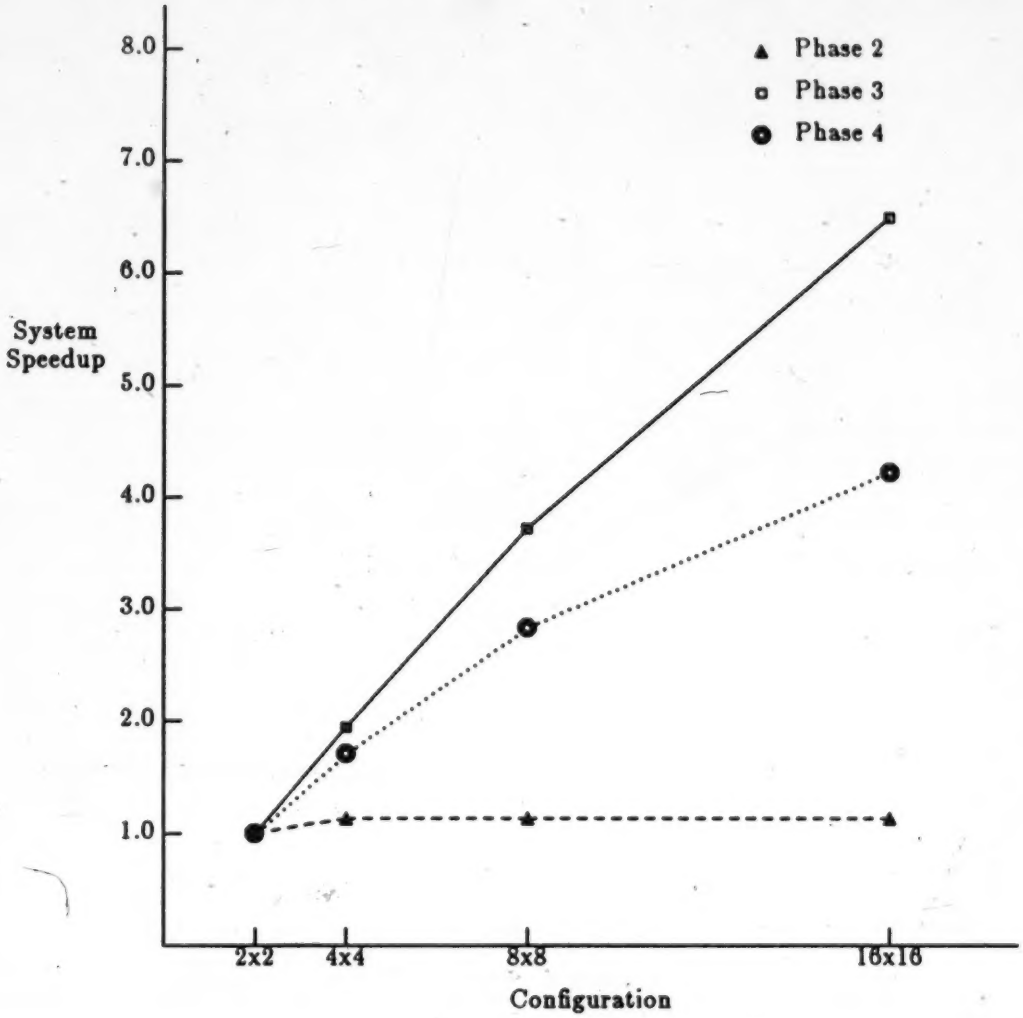


Figure 6.11 Speedup vs. Configuration For The Room (16 Proc. Per Node)

Resource	Configuration			
	2 X 2	4 X 4	8 X 8	16 X 16
CPU	80.30	45.43	22.72	11.37
Pixel Gun	2.76	1.56	0.78	0.39
Memory	26.91	15.82	8.02	4.04
A-bus	12.39	7.00	3.52	1.76
B-bus	17.45	10.20	5.18	2.60

Table 6.2 Percent Utilization of Phase 2 Resources

To test this, a polygon splitting program was written which "chopped" the polygons into smaller pieces.‡ Unfortunately, the room scene could not be split as much as desired since it would increase data requirements for the simulation and exceed virtual memory space allotted to users.• Despite this limitation, there was a great improvement in performance.

Figure 6.12 shows speedup obtained with a more uniform work distribution. Performance gains are particularly evident in the 16 and 32 processor cases. Speedup is examined for 16 processors per node and is plotted for each phase in Figure 6.13. Phase 2 now has an impressive increase in performance but only marginal gains are made for phases 3 and 4. Deficient speed-up in these phases can again be attributed to poor workload distribution.

The problem in Phase 4 is the work allocation scheme. For ease of implementation, it is assumed that only processors attached to processing node "n" can read fragment lists from multiport memory "n". Thus, processors at node 1 only read fragments from memory 1. If there are many more fragment lists in one memory module than in the others, the processors at the corresponding processing node will be over-worked while other processors stand idle.

This can be corrected by a better algorithm where processors are allowed to read from any module. So, in principle, nearly linear speedup is achievable in this phase.†

Phase 3, however, poses a more difficult problem. Non-uniform work distribution occurs because more fragments are "hitting" one module than others. This is no longer a problem with processor bandwidth but with storage speed.

---

‡ The characteristics of this data are given in the appendix.

• This occurred at 6.25 megabytes.

† This should be true for the range of processors that this study considers. It is recognized that if one module has many more fragments than the others, eventually all processors in the system will be reading from it. At this point memory becomes a bottleneck. But the time required to read a list is much shorter than processing it, so many processors can read from the same module without too much difficulty.

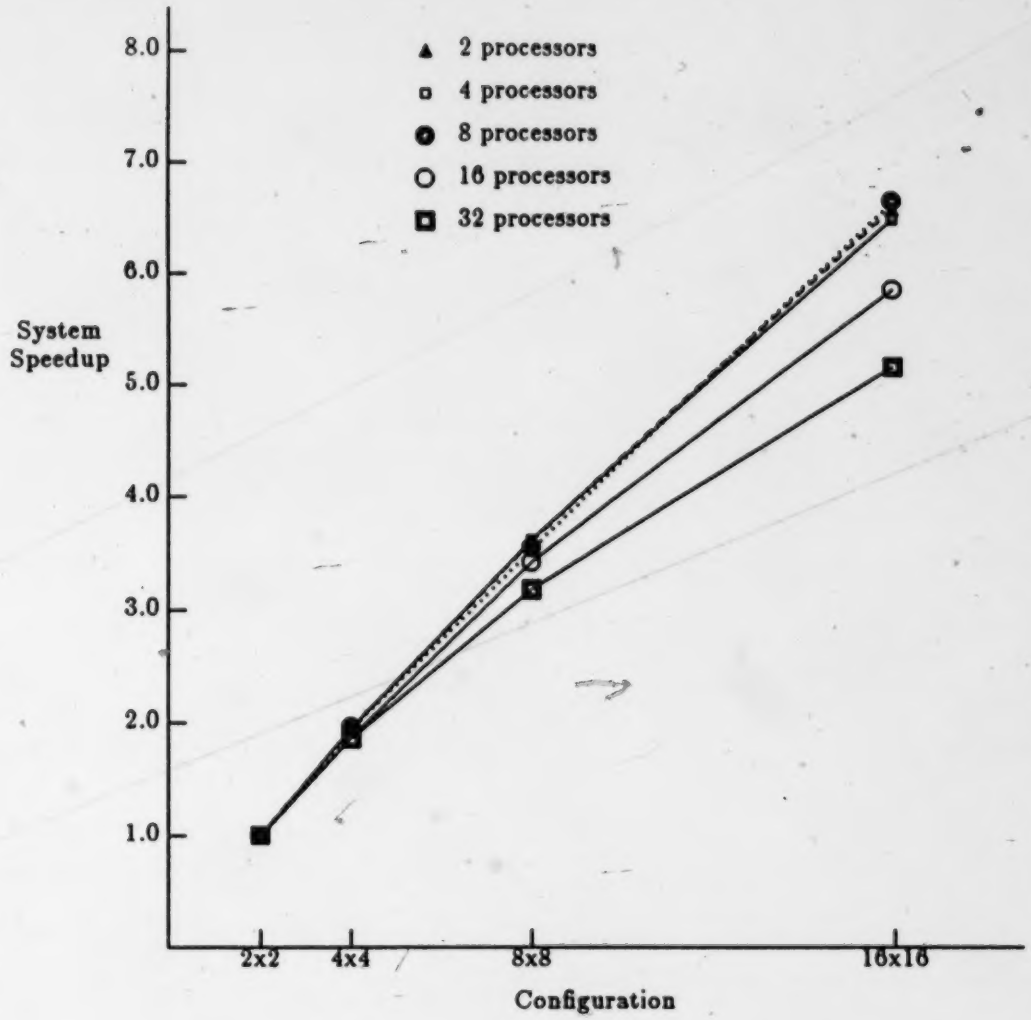


Figure 6.12 Speedup vs. Configuration For The Split Room

In random polygon scenes, fragments are uniformly distributed throughout the image. The room scene, on the other hand, has many vertical edges. This greatly increases the chances of storage imbalance in the system since all fragments along an edge fall into the same memory module. Expanding the system configuration exacerbates the problem.

Increased configuration sizes cause each memory module to sample smaller parts

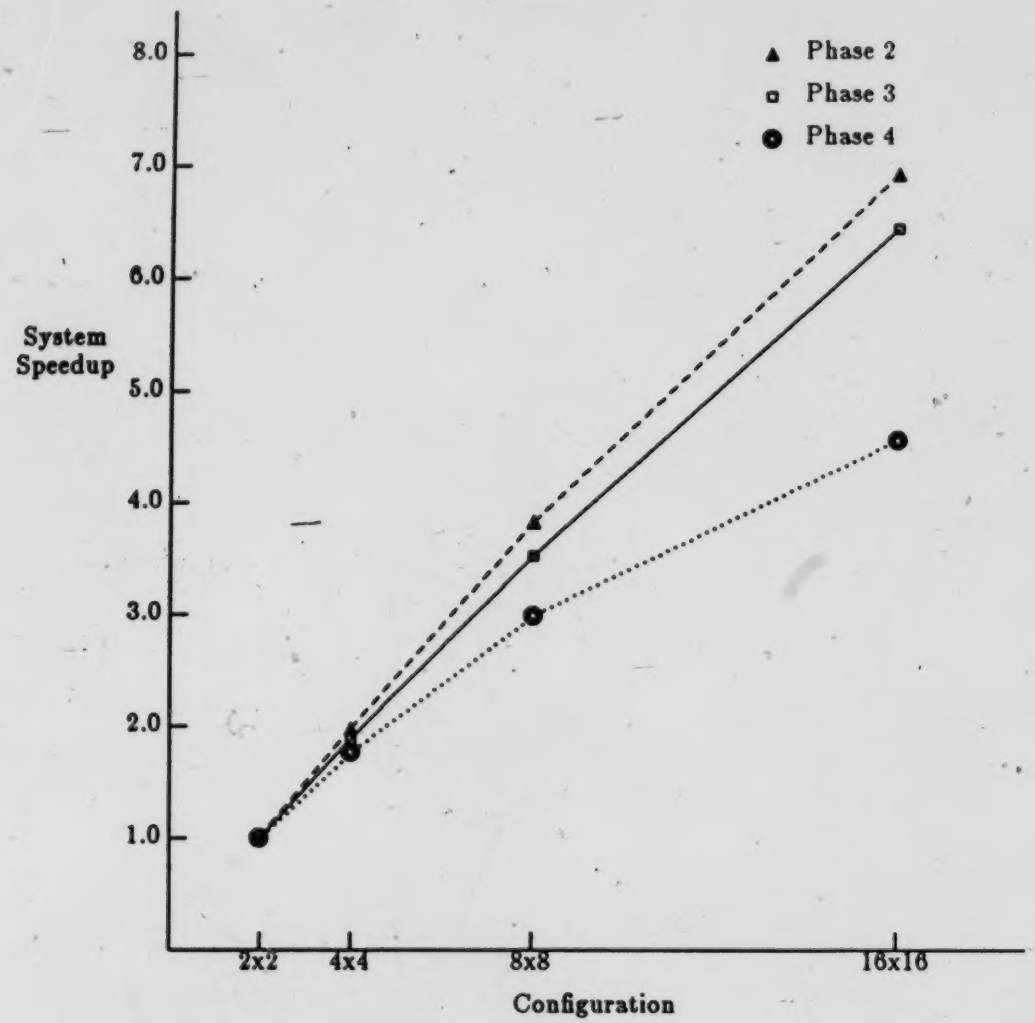


Figure 6.13 Speedup vs. Configuration For The Split Room (16 Proc. Per Node)

of the image, thus making it increasingly likely that one module has more fragments than another. The worst case occurs when there are as many modules as columns of pixels. Here, some modules may have no fragments while others are overwhelmed.

Since fragment dispersion is a function of the image, there is little that can be done. Splitting the polygons should have a modest effect since this will increase the number of fragments and might therefore reduce the relative imbalance between

modules. This dispersion problem is made more sensitive because of the slow fragment storage rates already observed in the previous experiments.

The major problems with scaling a system involve workload distribution. This is expected since all resources are added in direct proportion so balance between resources is maintained ie. the ratio between processors and busses stays constant. Workload distribution problems in Phase 4 are easily solved by changing the work allocation algorithm.

Better work allocation in Phase 2 is also easily achieved. The modeling system could solve this problem by insuring that polygons are uniformly small. The architecture could also achieve this by automatically splitting large polygons and storing them back into the host interface. This latter approach is more flexible and reduces the host's workload.

The only real obstacle to achieving nearly linear speedup is non-uniform dispersion of fragment data. This is not as severe a problem as exceeding a resource's capacity since this leads to minor degradation in only one of the four phases, also, not all images have serious dispersion difficulties.† A remedy to this problem would be to interleave the memories on the lowest order bits of both the x and y address. This would partition the screen image like a checker board among all the buffers, thus spreading workload from vertical edges more evenly.

† Images with vertical edges are more likely to have non-uniform dispersion and vertical edges are rare in applications such as flight simulators.

6.3. System Bottlenecks

As pointed out, there is ample evidence that the multiport memories are a bottleneck in the third phase of the algorithm. There is also good evidence that the microprocessor CPU is a bottleneck in the other phases. To confirm that CPU speed is a problem, all CPU time parameters were cut in half and a simulation was run using random polygon sets. If system performance is doubled by increasing CPU speed, then the microprocessor would be confirmed as a bottleneck.

Figure 6.14 gives a plot of the "normal" CPU divided by fast CPU times. Phase 2 and Phase 4 performance is evidently limited by the CPU speed. Phase 4 however, suffers a slight decline in performance between 16 and 32 processors per node. This indicates that another factor, such as B-bus contention, becomes more important. Phase 3 starts out CPU bound but the multiport memory is a bottleneck between 4 and 8 processors. Microprocessor speed is clearly a limiting factor, especially in Phase 2.

When CPU utilization is examined, it becomes clear that most of the time is spent in either initializing polygons or in initializing scanlines. Initializing a polygon and a scanline take 173 microseconds and 116 microseconds respectively.† But the ratio of time spent between these activities is data dependent. In Test Set #2, 14% of CPU time is spent in initializing polygons and about 83% in initializing scanlines — this results from the bigger polygons found in Test Set #2. Typically, time spent between these activities is more evenly balanced — the plant scene spends 47% in polygon initialization and 38% in scanline initialization. This data is summarized in Table 6.3.

† Appendix 2 gives other simulation parameters.

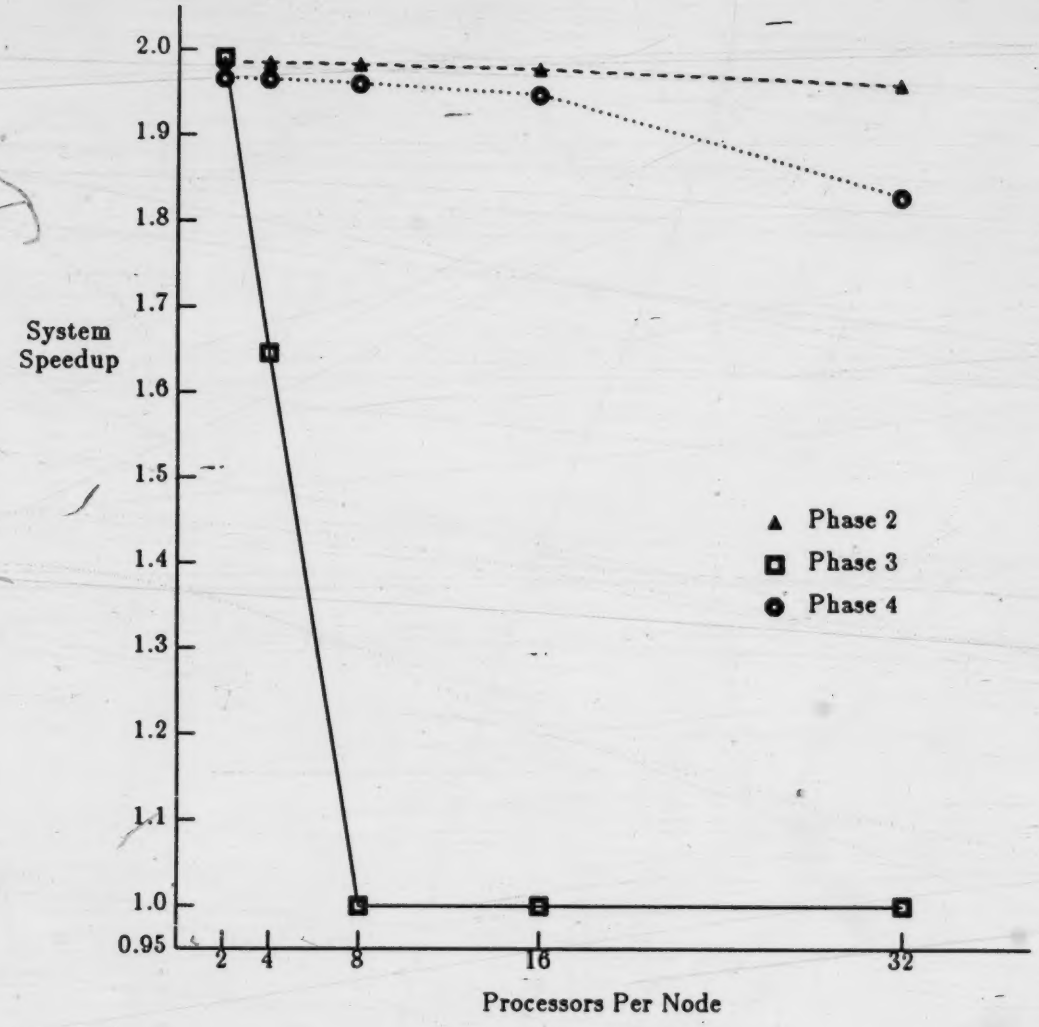


Figure 6.14 Normal CPU Times Over Fast Times vs. Processors (4-by-4)

Activity	Data Set			
	Test Set 1	Test Set 2	Room Scene	Plant
Polygon Initialization	51.2	14.3	53.6	47.4
Scanline Initialization	41.1	82.6	31.3	38.3

Table 6.3 Percent Activity For Phase 1 In The 4-by-4 Configuration

6.4. Data Dependencies

It was noted earlier that fragments cause much work in Phase 3. The amount of fragment generation depends on how much aliasing occurs in the scene. It was therefore decided to investigate the dependence of processing time on aliasing in a scene.

Unfortunately, it is difficult to separate some factors, like aliasing, from others — such as polygon size. This difficulty makes performance dependencies hard to determine. In the present case, high aliasing generally means low screen coverage, and low amounts of generated data. This causes a wide variance in time taken to generate a scene. Low aliased pictures take 213 milliseconds to render and have over 30 overlaps per pixel. Highly aliased images take 51 milliseconds and have less than one overlap.

To factor out these results, the time per partial result is plotted against percent aliasing (Figure 6.15).† An inspection of the graph shows a roughly linear dependence of processing time per partial results on percent aliasing. This relationship reflects the composition of the rendered image. At low aliasing, there are few fragments and more whole pixels produced. One would expect, at a lower bound of zero aliasing, that the time is determined by memory interleaving and rejection speed. This lower bound is 50 nanoseconds for an interleaving of four multiport memories and a rejection time of 200 nanoseconds.‡

An upper bound is also likely for 100% aliasing since each generated fragment takes a fixed amount of time to produce, store, and merge. The graph should therefore resemble an "S" shaped curve. Aliasing is clearly a factor in determining system performance.

The large amount of time to initialize a scanline produces a dependency on polygon orientation. Efficient production of whole pixels is dependent on how many pixels lie on an initialized scanline. This makes sense because the "true" cost of pro-

† Partial results are defined as the sum of the fragments and whole pixels produced.

‡ This represents the best case of 100% rejection, thus four pixels would be simultaneously rejected for a mean throughput of 50 nanoseconds each.

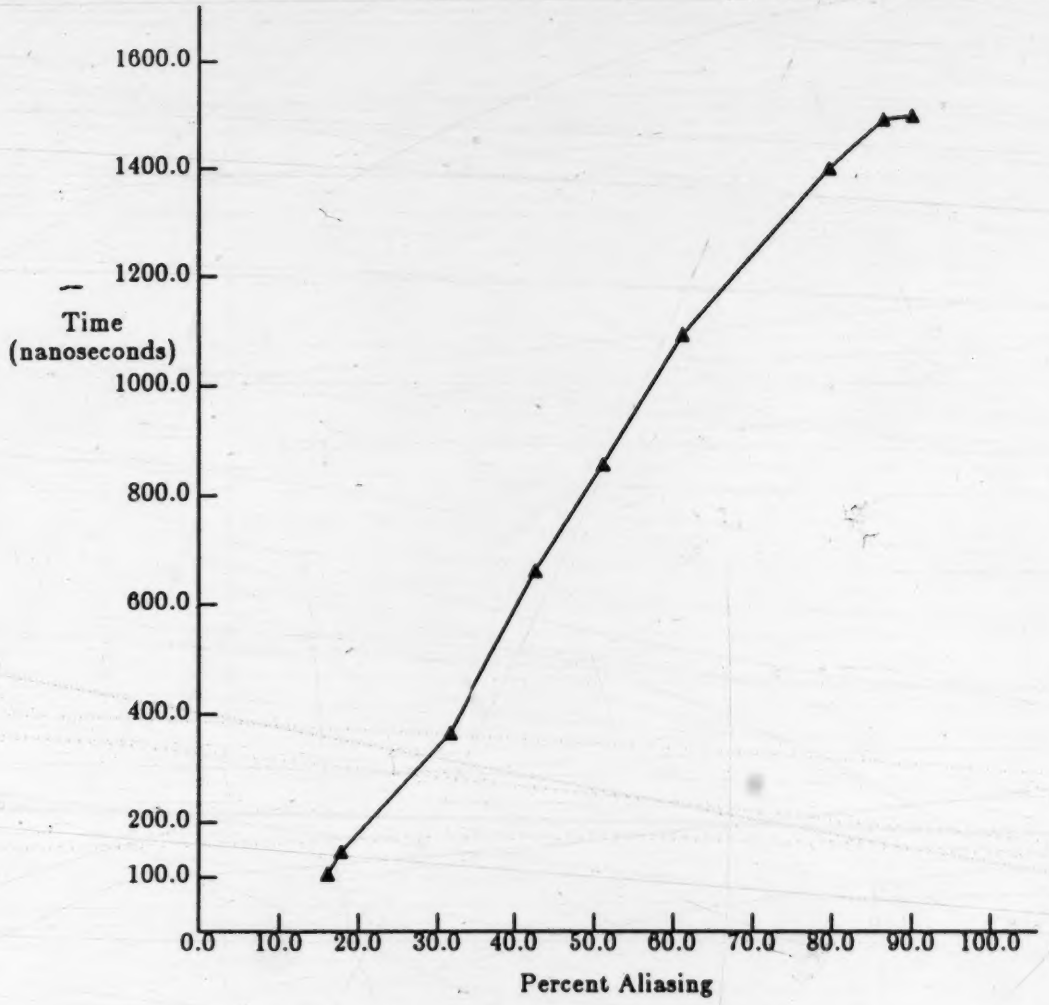


Figure 6.15 Time Per Partial Result vs. % Aliasing

ducing a whole pixel is the sum of the time taken by the pixel gun and the time to initialize the scanline. Producing pixels on short scan segments would therefore be very inefficient.

An experiment was constructed to observe the effect of very narrow, long rectangles on workload. The rectangles were randomly generated from a template having an aspect ratio of 15 to 1. There were 500 polygons in each file and they differed in their orientation. One file had all polygons oriented vertically, another had them oriented

horizontally, while a third file had all polygons randomly oriented. Table 6.4 gives the results of the experiment. There is almost a ten-fold ratio in the time taken to complete Phase 2 when comparing horizontal orientation to vertical orientation. This is caused by many more scanline initializations which must be done.

Phase	Polygon Orientation		
	Vertical	Random	Horizontal
2	611.6	428.2	71.8
3	84.0	82.6	64.1
4	55.6	65.9	49.8
Total Time	751.3	576.7	135.9

Table 6.4 Time (millisec.) to Render Asymmetric Polygons

There were other differences between the orientations. In Phase 3 the horizontal case was almost 20 milliseconds faster than the vertical case. The cause of this is likely a more uniform dispersion of fragments among the multiport memories. Thus all components of the system handle the workload evenly.

Phase 4 results show that randomly rotated polygons take longer to process than either vertical or horizontal. This is because there are more fragments produced by an edge which is not vertically or horizontally oriented. The slightly greater time taken by the vertical case is an indication of some multiport memories having more stored fragments than others — thus the effect of a load imbalance is observed.

6.5. Summary

Experiments with added processors show that the main obstacle to speed-up is the multiport memories in Phase 3. Fragment production speed appears to exceed their storage speed. Phase 2 resources are under-utilized because of insufficient whole pixel bandwidth.

Work allocation is found to be the biggest problem in scaling the architecture.

Phase 2 has workload distribution problems because of very large polygons, while Phase 4 has problems with the algorithm used to read fragment lists. Both of these obstacles are easily corrected.

Unfortunately, non-uniform distribution of aliasing in the scene causes some memory modules to be worked more than others and a degradation of speedup is experienced in Phase 3. This non-uniformity of aliasing is aggravated by vertical edges in the room scene. The problem can be corrected by interleaving the multiport memories on the last bits of both the x and y screen coordinates.

Aliasing is identified as a data characteristic causing greater workload in the system. In addition, whole pixel production efficiency is heavily dependent on the length of the scanline section being rendered.

Chapter 7

Conclusions and Future Research

7.1. Conclusions

This thesis concludes that the A-buffer algorithm can be adapted to a multiprocessor environment and therefore forms the basis for a good parallel antialiasing algorithm. The adapted algorithm produces good picture quality and gives realtime performance for many of the tested data sets.

Despite this, further work is needed to get the performance required for flight simulation.† Specifically, the memory bottleneck experienced in Phase 3 limits the system's speedup potential. There would be little trouble getting the required performance if linear speedup can be obtained with added processors. This bottleneck also makes the architecture performance more susceptible to data characteristics such as high amounts of aliasing.

Additional research should focus on finding a faster way of handling the generated fragments. At the cost of more overhead in Phase 4, one of the linked lists could be eliminated. There is also a possibility for more hardware support in this phase to help fragment storage.

Scaling the system gives much better speedup, but is not a flexible alternative and is more expensive. This expense is hard to justify since bus and memory resources are seriously underutilized in Phase 2.

A secondary problem is the lack of computational bandwidth in Phase 2. Ideally, this can be increased to balance processing bandwidth between the three phases. The objective is to balance throughput so that they all "bottleneck" with roughly the same number of processors. Boosting Phase 2 throughput would make the system less

† But the performance of this architecture is good enough for less demanding applications.

vulnerable to asymmetric polygons.

A secondary direction for research is to investigate ways to speed up Phase 2. In principle, more dedicated hardware can be applied since scanline initialization is a matter of linearly interpolating between two vertices. This is similar to the process used to find pixel colors between two polygon edges. VLSI assistance should be feasible, it may also be possible to integrate this into the pixel-gun. An even easier solution would be to use a faster microprocessor.

Once these obstacles are eliminated, particularly the Phase 3 bottleneck, nearly linear performance would be achieved and this system could be applied to flight simulation. The requirement that incoming polygons be non-intersecting, is not a liability in flight simulation since most of the objects are predefined and any additional expense in producing them is be easy to justify, especially considering that much more difficult and expensive means are currently used to achieve realtime for this same purpose.

This architecture has many desirable properties. Processor failure, for instance, causes only minor problems. The system can be run without the failed part and will experience only minor performance degradation. Therefore, the down-time for this system is low. Maintenance is also easier in this modular environment.

The architecture is flexible since different algorithms can be executed without hardware modification. It is possible, for instance, to render without shaded polygons — this greatly increases Phase 2 performance. The system can also be run in "vector mode" where thin, long polygons are used to represent vectors. This later change eliminates Phase 2 altogether since there are no whole pixels generated.

This thesis proposes two hardware aids useful on their own merits. The pixel-gun can be used in low-budget high-performance uni-processor systems. Performance of microprocessor based workstations, for instance, can be greatly increased with this co-processor. This VLSI chip also represents one of the first hardware devices proposed

for aiding the antialiasing process.

The arbitration scheme, if implemented on a chip, can be used as the building block for many different architectures, particularly those having a high degree of interconnection such as crossbar type systems. At the very least, this circuit makes it easier to implement multiport memory since ports can be added to the memory bus by simple modular connection to the bus.

7.2. Extensions and Further Work

Much work needs to be done to make this architecture a viable solution to the rendering problem. Transparency, for instance, is not implemented in the present system, although adding it is not likely to be a problem.† The simulated system has a frame buffer size of only 256-by-256 pixels. A higher quality production system would require four times as many pixels — namely 512-by-512. The appropriate system scale should be determined for such a system, although it is not expected that much more than twice the computational bandwidth would be required for the same performance.

Non-uniform distribution of fragments is the only real obstacle to obtaining linear speedup by scaling. An important improvement to this system would be to configure memory so that interleaving will occur on the lowest order bits of the y address in addition to the x address. This interleaving approach has already been tried in Fuchs' and Johnson's system[16], to overcome the same problem and merits investigation in this architecture.

Numerous hardware changes could be made to "tune" the system for optimal performance. One idea would be to store the Z depth value in 50 nanosecond RAM instead of 200 nanosecond RAM. Pixel rejection would be 4 times faster. The acceptance time would be only 250 nanoseconds instead of the 400 nanoseconds currently

† The A-buffer algorithm handles transparent bitmaps easily.

required. This would be a tremendous boost to memory performance in Phase 2 and 3.

Phase 4 performance could be improved with a VLSI "pixel merger". The merging operations are all simple since they require only logical operations with bitmaps. It should be relatively easy to build a device which takes depth-sorted fragment lists and produces pixels color as output.

Various changes could be made to the algorithm to improve performance. After Phase 2 polygon reading, polygons could be stored at each processing board instead of being written back into the host buffer. This would help relieve contention problems at the A-bus.

The capabilities of the system could be expanded to encompass clipping, scaling and transformation of polygons. One way to accomplish this would be to place a pipeline of "Matrix Engines" before each host buffer. With this scheme, the host only needs to specify polygon transformations, thus even more of the image generation process is off-loaded into the architecture.

There may be some potential for implementing Phong shading on the pixel-gun. This should be investigated as a means of obtaining higher quality images. Since the pixel-gun is many times faster than the CPU there should be no problems with accepting a slower (but higher quality) rendering scheme.

Further research needs to determine if this architecture could fill all the needs of a flight simulation environment. One concern is whether fast texture mapping is possible in this system. Texture mapping is common and desirable for realism in flight simulators.

Of course, there are numerous other practical details which need to be resolved. The pixel-gun and bus arbiter need to be designed and implemented for a prototype system. The frame buffer interface to the monitor also needs work.

This thesis takes only a small step towards creating a system for realtime image generation; there is much more work to be done before such a system can be built. But it should be remembered that "a journey of a thousand miles begins with a single footstep."

References

[1] Bechtolshiem, A. and Baskett, F., "High-Performance Raster Graphics for Microcomputer Systems", *Siggraph ACM*, Vol. 14, No. 3, July 1980, pp. 43-47.

[2] Locanthi, B., Object Oriented Raster Displays, *Proceedings of Caltech Conference on Very Large Scale Integration*, 1979, pp. 215-225.

[3] Bowen, B.A. and Buhr, R.J.A., *The Logical Design of Multiple Microprocessor Systems*, Prentice Hall, Englewood Cliffs, NJ, 1st edition 1980.

[4] Fussell, D. and Rathi, B.D., A VLSI-Oriented Architecture for Real-Time Raster Display of Shaded Polygons, *Proceedings of Graphics Interface 82*, 1982, pp. 373-380.

[5] Whelan, D.S., "A Rectangular Area Filling System Architecture", *Computer Graphics*, Vol. 16, No. 3, July 1982, pp. 356-362.

[6] Fiume, E., Fournier, A. and Rudolph, L., "A Parallel Scan Conversion Algorithm for A General-Purpose Ultracomputer", *Computer Graphics*, Vol. 17, No. 3, July 1983, pp. 141-150.

[7] Catmull, Edwin, "A Hidden Surface Algorithm with Antialiasing", *Computer Graphics*, Vol. 12, No. 3, August 1978, pp. 6-12.

[8] Crow, F.C., "The Aliasing Problem in Computer Generated Shaded Images", *Comm. ACM*, Vol. 20, No. 11, November 1977, pp. 799-805.

[9] Crow, F.C., "An Approach to Real-Time Scan Conversion", *AFIPS Conf. Proc.*, Vol. 48, 1979, pp. 157-163.

[10] Crow, F.C. and Howard, M.W., "A Frame Buffer System with Enhanced Functionality", *Computer Graphics*, Vol. 15, No. 3, August 1981, pp. 63-69.

[11] Crow, F.C., "A Comparison of Antialiasing Techniques", *IEEE Computer Graphics & Applications*, Vol. 1, No. 1, January 1981, pp. 40-48.

[12] Parke, F.I., "Simulation and Expected Performance of Multiple Processor Z-buffer Systems", *Siggraph ACM*, Vol. 14, No. 3, July 1980, pp. 48-56.

[13] Cioffi, G. and Velardi, P., "A Fully Distributed Arbiter for Multiprocessor Systems", *Microprocessing and Microprogramming*, Vol. 11, 1983, pp. 15-22.

[14] Broomwell, G. and Heath, J.R., "Classification Categories and Historical Development of Circuit Switching Topologies", *ACM Computing Surveys*, Vol. 15, No. 2, June 1983, pp. 95-134.

[15] Anderson, G.A. and Jensen, D.E., "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", *Computing Surveys*, Vol. 7, No. 4, December 1975, pp. 197-213.

[16] Fuchs, H. and Johnson, B.W., An Expandable Multiprocessor Architecture for Video Graphics, *Proceedings of the Sixth Annual Symposium on Computer Architecture*, 1979, pp. 58-67.

[17] Fuchs, H., Poulton, J., Paeth, A. and Bell, A., Developing Pixel Planes, A Smart Memory-Based Raster Graphics System, *Proceedings of the Conference on Advanced Research in VLSI*, January 1982, pp. 137-146.

[18] Niimi, H., Imai, Y., Murakami, M., Tomita, S. and Hagiwara, H., "A Parallel System for Three-Dimensional Color Graphics", *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 67-76.

[19] Christiansen, H. and Stephenson, M., *Movie.BYU Training Text*, University Press, Brigham Young University, 1983-84 Edition.

[20] Gouraud, Henri, "Continuous Shading of Curved Surfaces", *IEEE Trans. on Computers*, Vol. 20, No. 6, 1971, pp. 623-629.

[21] Sutherland, I.E., Sproull, R.F. and Schumacker, R.A., "Sorting and the Hidden-Surface Problem", *AFIPS Conf. Proc.*, Vol. 42, June 1973, pp. 685-693.

[22] Sutherland, I.E., Sproull, R.F. and Schumacker, R.A., "A Characterization of Ten Hidden-Surface Algorithms", *Computing Surveys*, Vol. 6, No. 1, March 1974, pp. 1-45.

[23] Foley, J.D. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

[24] Jackson, J.H., "Dynamic Scan-Converted Images With a Frame Buffer Display Device", *Siggraph ACM*, Vol. 14, No. 3, 1980, pp. 163-169.

[25] Clark, J.H., "The Geometry Engine: A VLSI Geometry System for Graphics", *Computer Graphics*, Vol. 16, No. 3, July 1982, pp. 127-133.

[26] Acquah, James, Foley, James, Siebert, John and Wenner, Patricia, "A Conceptual Model of Raster Graphics Systems", *Computer Graphics*, Vol. 16, No. 3, July 1982, pp. 321-328.

[27] Thurber, K.J. and Wald, L.D., "Associative and Parallel Processors", *Computing Surveys*, Vol. 7, No. 4, December 1975, pp. 215-255.

[28] Carpenter, L., "The A-buffer, an Antialiased Hidden Surface Method", *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 103-108.

[29] Kaplan, M. and Greenberg, D.P., "Parallel Processing Techniques for Hidden Surface Removal", *ACM Computer Graphics*, Vol. 13, No. 2, August 1979, pp. 300-307.

[30] Satyanarayanan, M., *Multi-Processors: A Comparative Study*, Prentice-Hall, New Jersey, 1980.

[31] Whitton, M.C., "Memory Design for Raster Graphics Systems", *IEEE CG&Appl.*, Vol. 4, No. 3, March 1984, pp. 48-65.

[32] Enslow, P.H., "Multiprocessor Organization - A Survey", *Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 103-129.

[33] Baecker, R., "Digital Video Display Systems and Dynamic Graphics", *Siggraph ACM*, Vol. 13, No. 2, August 1979, pp. 48-56.

[34] Gemballa, R. and Lindner, R., "The Multiple-Write Bus Technique", *IEEE C. G. & Appl.*, Vol. 2, No. 7, September 1982, pp. 33-41.

[35] Sproull, R.F., Sutherland, I.E., Thompson, Alistair, Gupta, Satish and Minter, Charles, "The 8 by 8 Display", *ACM Transactions on Graphics*,, Vol. 2, No. 1, January 1983, pp. 33-56.

[36] Weinberg, Richard, "Parallel Processing Image Synthesis and Anti-Aliasing", *Computer Graphics*, Vol. 15, No. 3, August 1981, pp. 55-61.

[37] Weinberg, Richard, *An Architecture for Parallel Processing Image Synthesis with Anti-aliasing*, PhD Thesis, University of Minnesota, 1982.

[38] Gupta, S., Sproull, R.F. and Sutherland, I.E., "A VLSI Architecture for Updating Raster-Scan Displays", *Computer Graphics*, Vol. 15, No. 3, August 1981, pp. 71-78.

[39] Whitted, T., Hardware Enhanced 3-D Raster Display System, *Proceedings of the 7th Man-Computer Communications Conference*, 1981, pp. 349-356.

[40] Giloi, W.K., *Interactive Computer Graphics: Data Structures, Algorithms, Languages*, Prentice Hall, Englewood Cliffs, NJ, 1978.

[41] Newman, W.M. and Sproull, R.F., *Principles of Interactive Computer Graphics*, Second Edition, McGraw-Hill, 1979.

[42] Paker, Y., *Multi-Microprocessor Systems*, Academic Press (London), 1983.

Appendix A1

Simulation Methodology

1.1. The Experimental Design

1.1.1. The Simulation

The experiments supporting this thesis are founded on many pieces of software. The simulation itself is composed of about 7000 lines of code written in "C" and is data driven. It is structured to execute the proposed algorithms while simulating architectural performance. Thus, whenever a polygon initialization event takes place, a polygon is actually initialized. Many purposes are served by structuring the simulator in this fashion.

If the simulated architecture is to be tested for data dependencies, real data must be used. The speed of execution depends on the algorithm and data so the easiest way to observe dependencies is to assign invariant segments of code a constant execution speed and let the data determine how often and when code is executed.

Another reason for taking this approach is to verify simulation correctness. If, for instance, the simulation is modeling bus transfers incorrectly, the resulting image would be distorted since the correct execution of the graphics algorithm is tied to the event simulation.

New graphics algorithms were used so they needed to be verified. It was also important to see the image quality resulting from the algorithms since architecture and algorithm speed means nothing if image quality were poor. All the above reasons guided the design of the simulator.

1.1.2. MOVIE.BYU

Since generation of "real" scenes is painstaking work, it is best to use pre-existing image data. Unfortunately, data has to be preprocessed to do the lighting and scaling calculations before rendering. This in itself, is an enormous amount of work. An additional problem is finding a standard for comparing generated images. As it turns out, the Movie.BYU graphics package provides a solution to both problems.

Fortunately there is a large amount of image data available for the Movie.BYU package and there was also access to its source code. Movie.BYU was modified to output polygons which are scaled and have lighting values. This is an ideal solution since Movie.BYU images can be compared with the "A-buffer" images. The only difference between the Movie.BYU and A-buffer images is in the rendering since lighting and scaling are identical.

1.1.3. Polygon Generator

It was also necessary to write a polygon generator to have greater control over polygon data characteristics used in the simulation. The polygon generator, as implemented, takes "templates" or prototypical polygons and randomly scales, rotates and translates them to create a file of polygons with randomly assigned depth and color values.

The generator takes up to 10 different prototypes and mixes them with a weighted probability taken from the user. Control over data characteristics is accomplished by changing the supplied templates and by adjusting their probability of inclusion. This means of control is not exact, and not all parameters can be controlled independently of others.

1.1.4. Other Software

Many other utilities were written to "massage" data for experimentation. A scrambling program was written to scramble the order of polygon data to determine if ordering has an effect. A "smoothing" program was written to average the colors of polygon vertices — smoothing is not done by the modified Movie.BYU package. A polygon splitter was written to split polygons over a specified size. Some debugging tools were also implemented to help isolate problem causing data.

1.2. The Level of Simulation

The simulator is a discrete event simulator that executes the graphics algorithm when scheduled events arise. "Events" are defined as system actions which can not be interrupted. Thus events are indivisible actions taken by the system, like initializing a polygon. Since event can not be interrupted it is safe to assign them a benchmarked time. This time is used to update the event clock when an event is executed.

Unfortunately, a few events can not be assigned one completion time. Some events, such as preparing a scanline, could take many different steps. In these cases loops or branches within the routines have to be benchmarked individually. This is done for only a few routines since most could be assigned a constant time. This variable execution time causes some problems with the simulation, and is described below.

Processor states are modeled by reserving storage for intermediate variables used in the graphics calculations. When a particular phase of the algorithm is executed this processor "image" is updated to reflect the computation.

Since events have variable, and unpredictable, completion times, events are performed to determine these times. Unfortunately this changes the processor state before the event is scheduled to complete and causes inconsistencies in the simulation. The solution to this is in performing the calculations on a "reserve processor image".

This allows prediction of event completion times without affecting processor state. When an event completes (according to the simulation clock), the reserved state is transferred to the processor image.

1.2.1. CPU and Pixel Gun States

The modeled microprocessor and pixel-gun states are listed below. These states are dependent on polygon characteristics and state of the

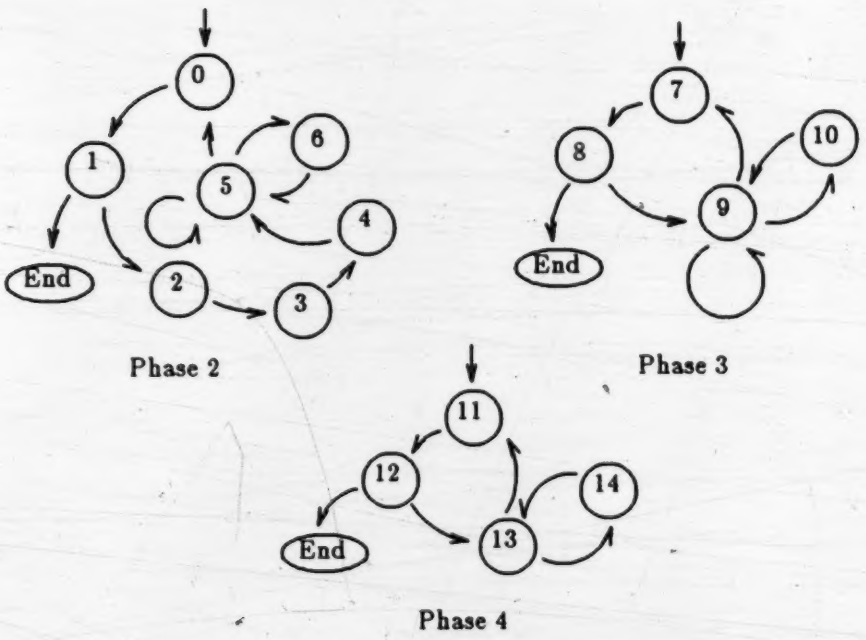


Figure A1.1 Microprocessor State Diagram

busses. Figure A1.1 gives the state transition diagram of the microprocessor CPU and Figure A1.2 is the state diagram for the pixel-gun. Numbers for these states are explained in Tables A1.1 and A1.2.

State transitions in Phase 2 are the most complicated of the three modeled phases. As in all three phases, the CPU starts idle waiting to use the bus (State 0). When the bus is free, reading starts (State 1) and the processor may either stop (if it reads a termination signal) or it starts calculating dz values for the current polygon

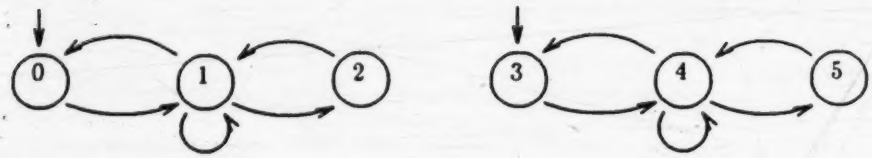


Figure A1.2 Pixel Gun State Diagram

(State 2). Initialization follows (State 3) and the CPU is temporarily blocked until the pixel-gun is free (State 4). After this the CPU initializes scanlines (State 5) and repeats until the last scanline is initialized. Occasionally the CPU becomes blocked waiting for the pixel-gun (State 6), but this rarely happens in practice.

Phase 3 follows a similar sequence but omits polygon initialization and calculation of dz values. The sequence starts with waiting to access the A-bus (State 7) and then goes to a "reading" state (State 8). The sequence ends if the last polygon is read, or it proceeds to edge initialization (State 9). Edge initialization may become blocked if the pixel-gun is busy (State 10), otherwise it cycles until the last edge is read whereupon it proceeds into the wait state for the A-bus.

The last modeled phase (Phase 4) is similar to the above, except that the CPU is waits for different system resources. It starts by waiting to read a fragment list from the B-bus (State 11). It stops upon reading a termination signal, otherwise the list is merged and it attempts to store (State 13). The CPU goes to the blocked state if the buffers are full (State 14), otherwise it attempts to read another pixel list. These states are listed in Table A1.1.

The pixel-gun states are simple; either a pixel-gun is waiting for the initialization parameters (States 0 or 3), or it is busy producing data (States 2 or 5). It is also possible for the pixel-gun to be blocked by full storage buffers. These states are listed in Table A1.2.

Phase	State Number	State Description
Phase 2	0	idle, waiting for a polygon
	1	Busy, reading a polygon
	2	Busy, calculating dx values and storing them
	3	Busy, initializing before scanline processing
	4	Idle, waiting for the pixel-gun to finish
	5	Busy, calculating scanline parameters
	6	Idle, blocked in "ready_scan" waiting for the pixel-gun to finish
Phase 3	7	idle, waiting to read a polygon
	8	Busy, reading a polygon
	9	Busy, executing "fragen_init"
	10	Idle, waiting for the pixel-gun to finish
Phase 4	11	idle, waiting to read a fragment list
	12	Busy, reading a pixel list
	13	Busy, sorting and merging
	14	Idle, blocked on a pixel store

Table A1.1 Simulated CPU States

Phase	State Number	State Description
Phase 2	0	Idle, waiting to make a pixel
	1	Busy, making pixels
	2	Idle, blocked on storing pixels
Phase 3	3	Idle, waiting to make fragments
	4	Busy, making fragments
	5	Idle, blocked on storing fragments
Phase 4	The pixel-gun is idle during Phase 4.	

Table A1.2 Simulated Pixel Gun States

This simulation gathers statistical data indicating utilization of memory, CPU, A-bus, and B-bus. There is also a tabulation of the percentage time spent in each phase of operation.

All data was collected every 1333 nanoseconds of simulated time. This rate was fast enough to give a fairly accurate picture of system operation. It was also chosen to be an odd number that would not coincide with any regular frequency of system operation. This should alleviate concerns that sampling rate aliases events in the system.

1.3. The Data

Data used in the experiments is derived from different sources. Natural scene data is taken as output from a modified Movie.BYU graphics package. As well, random polygons are generated. There is a qualitative and quantitative difference between these two data types which should be pointed out.

1.3.1. Distribution of Complexity

Complexity is uniformly distributed throughout the frame buffer in a randomly generated scene. Natural scenes, on the other hand, tend to have complexity localized. Modeled objects such as vases may contain hundreds of tiny polygons while, in the same scene, there may exist large walls modeled with a single polygon.

Complexity in a real scene is nearly always visible. The person modeling a scene almost never put a complex object such as a vase behind a wall. Randomly generated polygons have as much complexity hidden as there is visible. This makes a difference in the rejection rates of fragments in the two different data sets. It is probable that given a scene and a random data set with the same amounts of aliasing, the natural scene will have a lower fragment rejection rate.

Visibility of complexity also reduces the number of pixel overlaps in a scene. Randomly generated data should have more pixel overlaps and therefore a higher whole data rejection rate.

Another important difference is how objects are modeled in a natural scene. Polygons in objects tend to have "butting" edges. A curved surface, for instance, is modeled with many butting polygons. Randomly generated polygons have no butting contacts. This makes a difference in the depth of generated fragment lists.

Since a shared edge causes generation of two fragments for each pixel the edge covers, fragment lists in a natural scene tend to be much longer and aliasing is more

localized. In a random scene fragment lists are shorter and more uniformly distributed.

1.3.2. Heterogeneity of Polygon Size

Random polygons are generated by scaling a "template polygon" with a normally distributed scaling factor. This tends to produce similar sized polygons. But the distribution of polygon size in the room scene does not follow a normal distribution — there are a great number of small polygons with only a few large polygons. This has implications on workload distribution and is noted in Chapter 6. Table A1.3 gives some of the characteristics of the data sets.

Characteristics	Data Set				
	Room	Plant	Test 1	Test 2	Split Room
Total Polygons	1661	1585	1625	1625	2244
Total Edges	6535	6116	6335	6341	8866
Average Edges per Polygon	3.94	3.86	3.90	3.90	3.95
Average Pixels per Edge	7.61	6.32	6.56	16.5	7.06
% Frame Covered	69.7	21.6	40.8	94.7	69.7
Relative Aliasing (% of stored pixels)	20.7	46.4	90.1	30.4	26.8
Total Aliasing (% of Frame)	16.34	11.88	37.67	74.92	21.76
Visible Aliasing (% of Frame)	14.42	10.02	36.73	28.78	18.65
Average Overlaps	1.65	2.684	0.685	4.405	1.65

Table A1.3 Data Characteristics of Experimental Data Sets

1.4. Simulation Assumptions and Level of Modeling

1.4.1. Simulation Parameters

Parameters used in this simulation are derived by various means. Bus and memory speed are part of the architectural definition. Other parameters are a function of the algorithm and processing speed. Since the simulation is designed to test the architecture and not a particular implementation, it is felt that exact benchmarking is not required to derive parameters. If the numbers are reasonable and proportionately correct, then simulation results should not be biased. The simulated architecture

should show the same sensitivity and trends to changes in the data and to various configurations.

To expound on this point, the reader should realize that the emphasis of this study is to observe architectural behavior with changing configuration and data. It is not intended to discover how fast it is, although "ball-park" estimates are made. Since benchmarking only involves microprocessor execution time, the derived parameters should be proportionately correct. Thus one can validly argue that a microprocessor exists or can be constructed to give roughly the same performance as specified by the parameters.

A further, retrospective, justification of this method is the results obtained for various phases. The microprocessor is vastly slower than other system components in Phases 2 and 4 so considerable latitude can be given to CPU execution speed. One can, for instance, greatly increase CPU speed before it ceases to be a bottleneck. The bitslice processor bandwidth in Phase 3 is still a bottleneck since the benchmarked parameters are roughly correct. Also, these parameters are likely to be a lower performance bound for any realistic system.

With this in mind, a rough benchmarking was done to approximate the time taken to execute the algorithms in fixed point arithmetic. Segments of code were reduced to essentials and benchmarked on the VAX 11/780 to approximate execution times on the microprocessors. Time taken to execute 100,000 iterations of this code was divided by 100,000 to give the approximate value. This probably gives a lower bound to processor performance since an actual implementation would use hand coded assembler, and would likely run on a processor of the Motorola 68000 family. The greater number of registers available could make it possible to store nearly all the data on the processor thus reducing memory fetches to instructions only. On the 68020, with an instruction cache, this could be an impressive performance gain.

Parameters for the bitslice processor had to be estimated since there was no existing processor that could give an approximate benchmark. It was felt, however, that a high speed processor could be built. An examination of the 1982 Motorola MECL guide shows that circuits exist that do comparisons in under 3.6 nanoseconds. A 4-bit ALU takes 10.8 nanoseconds (in the worst case) and RAM is available which operates in under 30 nanoseconds. Since processor operations are simple, the processor can be built with this technology. Given this assumption, the limiting factor in processor speed is the time to access common RAM (200 nanoseconds). Therefore, time to complete the bitslice processor algorithms are estimated by the number of data fetches and stores made to RAM.

1.4.2. Definition of Statistical Parameters Collected

Most data characteristics that were collected are self-explanatory, however, others are a little more subtle. Below is a list of collected parameters requiring further explanation in order to properly interpret them. These definitions correspond to the simulation output in the following appendix.

1.4.2.1. Total Number of Pixels Covered:

This represents the number of pixels where an attempt was made to store either a whole pixel or a pixel fragment.

1.4.2.2. Total Number of Whole Pixels On the Screen:

This is a count of all the locations in memory where at least one whole pixel was stored. The same location may also contain a fragment.

1.4.2.3. Total Number of Whole Pixels Stored:

This statistic counts the number of times pixels were stored. This also includes whole pixels that may be subsequently over written.

1.4.2.4. Total Number of Fragments

The number of times fragments have been accepted by the Z-buffer processor for storage (see definition of "Total Number of Whole Pixels Stored.").

1.4.2.5. Total Number of Antialiased Pixels On The Screen

This is the count of all pixels where at least one fragment must be merged.

1.4.2.6. Percentage of Frame Covered:

The number of pixel locations where either a fragment or a whole pixel was stored, divided by the number of pixels on the screen, and multiplied by 100%.

1.4.2.7. Whole Pixels (% of Frame):

This represents the number of locations where a whole pixel was stored, divided by the number of pixels on the screen, and multiplied by 100%. Note that pixel fragments may also be stored into the same memory locations.

1.4.2.8. Aliased Pixels (All) (% of frame):

The number of unique pixel locations where a fragment store was attempted, divided by the number of pixels in the frame.

1.4.2.9. Aliased Pixels Exposed (% of frame):

The number of locations where a fragment was stored, divided by the number of pixels on the screen.

1.4.2.10. Aliased Pixels (% of Stored Pixels):

The number of locations where a fragment was stored, divided by the number of locations where anything was stored.

1.4.2.11. Average Number of Pixels per Edge:

The number of fragments generated, divided by the number of edges.

1.4.2.12. Average Number of Pixels per Polygon:

The total number of pixel locations where either a fragment was stored or a whole pixel divided by the number of polygons.

1.4.2.13. Average Number of Overlaps per Pixel:

The total number of fragments and whole pixels generated, divided by the total number of locations where either a fragment or a whole pixel was stored.

Appendix A2

Sample Simulation Output

```
################################################################################
SIMARCH PERFORMANCE OUTPUT FILE - run initiated on Wed Aug 20 13:34:59 1986
INPUT TAKEN FROM FILE: plant
################################################################################
 ---------------------- CPU States Summary ----------------------
 ---------------------- Phase 1 CPU States ----------------------

 0 - idle, waiting for a polygon
 1 - busy, reading a polygon
 2 - busy, calculating dz values and storing them
 3 - busy, initializing polygon parameters
 4 - idle, after poly initialization blocked by the pixel generator
 5 - busy, calculating scanline parameters for the pixel generator
 6 - idle, after scanline preparation blocked by the pixel generator

 ---------------------- Phase 2 CPU States ----------------------
 7 - idle, waiting for a polygon in the fragment phase
 8 - busy, reading a polygon in the fragment phase
 9 - busy, initializing the fragment generation algorithm
 10 - idle, waiting for the VLSI processor to finish

 ---------------------- Phase 3 CPU States ----------------------
 11 - idle, waiting to read a pixel list
 12 - busy, reading a pixel list
 13 - busy, sorting and merging
 14 - idle, blocked on a pixel store

 ------------------------ Pixel Processor State Table -----------------
 ---------------------- Phase 1 Pixel Processor States ---------------
 0 - idle, waiting to make pixels
 1 - busy, making pixels
 2 - idle, blocked on a pixel store

---------------------- Phase 2 Pixel Processor States ---------------
 3 - idle, waiting to make fragments
 4 - busy, making fragments
 5 - idle, blocked on storing fragments
```

$$$$$$$$$$$$$$$ Phase 1 System Performance $$$$$$$$$$$$$$$

CPU STATE SUMMARY

CPU STATE	PERCENTAGE TIME
0	43.9060
1	0.1808
2	3.5979
3	28.9255
4	0.0000
5	23.3880
6	0.0019
7	0.0000
8	0.0000
9	0.0000
10	0.0000
11	0.0000
12	0.0000
13	0.0000
14	0.0000

PIXEL PROCESSOR STATE SUMMARY

PIXEL PROCESSOR STATE	PERCENTAGE TIME
0	99.6263
1	0.3737
2	0.0000
3	0.0000
4	0.0000
5	0.0000
6	0.0000

Average CPU busy: 56.092111
Average Pixel Processor busy: 0.373687
Average Memory Module busy: 8.056121

A Bus busy: 14.807060
B Bus busy: 4.690435

Maximum Observed A buffer length: 2
Maximum Observed B buffer length: 2
Maximum Observed Multiport buffer length: 2

Average A buffer length: 0.015608
Average B buffer length: 0.001519
Average Multiport buffer length: 0.021314
Total number of observations: 12776

Phase 1 complete after 17040153 nanoseconds
##

$$$$$$$$$$$$$$$$$$$$ Phase 2 System Performance $$$$$$$$$$$$$$$$$$$$

CPU STATE SUMMARY

CPU STATE	PERCENTAGE TIME
0	0.0000
1	0.0000
2	0.0000
3	0.0000
4	0.0000
5	0.0000
6	0.0000
7	26.4133
8	0.2626
9	23.3142
10	50.0099
11	0.0000
12	0.0000
13	0.0000
14	0.0000

PIXEL PROCESSOR STATE SUMMARY

PIXEL PROCESSOR STATE	PERCENTAGE TIME
0	0.0000
1	0.0000
2	0.0000
3	33.8571
4	1.2491
5	64.8938
6	0.0000

Average CPU busy: 23.576753
Average Pixel Processor busy: 1.249127
Average Memory Module busy: 96.901615

A Bus busy: 8.402002
B Bus busy: 96.831103

Maximum Observed A buffer length: 1
Maximum Observed B buffer length: 64
Maximum Observed Multiport buffer length: 64

Average A buffer length: 0.006249
Average B buffer length: 49.824667
Average Multiport buffer length: 35.418095

Total number of observations: 18082

Phase 2 complete after 41143303 nanoseconds
##

$$$$$$$$$$$$$$$$$$$$ Phase 3 System Performance $$$$$$$$$$$$$$$$$$

CPU STATE SUMMARY

CPU STATE	PERCENTAGE TIME
0	0.0000
1	0.0000
2	0.0000
3	0.0000
4	0.0000
5	0.0000
6	0.0000
7	0.0000
8	0.0000
9	0.0000
10	0.0000
11	0.7382
12	8.3088
13	90.9530
14	0.0000

PIXEL PROCESSOR STATE SUMMARY

PIXEL PROCESSOR STATE	PERCENTAGE TIME
0	0.0000
1	0.0000
2	0.0000
3	100.0000
4	0.0000
5	0.0000
6	0.0000

Average CPU busy: 99.261834
Average Pixel Processor busy: 0.000000
Average Memory Module busy: 25.085368

A Bus busy: 0.000000
B Bus busy: 30.023903

Maximum Observed A buffer length: 0
Maximum Observed B buffer length: 2
Maximum Observed Multiport buffer length: 2

Average A buffer length: 0.000000
Average B buffer length: 0.019616
Average Multiport buffer length: 0.065189

Total number of observations: 5857

Phase 3 complete after 48949896 nanoseconds
##

System Configuration Parameters:

Frame Size: 256 X 256 pixels
Number of Nodes: 4
Number of Processors per node: 32

Pixel Resolution: 256
Number of Memory Modules: 4
Maximum Buffer sizes: 64

Red Pixel Starting Value: 20
Blue Pixel Starting Value: 20
Green Pixel Starting Value: 20

Timing Parameters:

All the following numbers are for times to complete the described task
Reading a whole vertex: 1200 nsec
Traversing a vertex description: 500 nsec
Preparing a polgon for pixel production: 173334 nsec
Calculating the dz/dy, and dzdx values: 49500 nsec
Storing the dz values: 400 nsec
Checking if any polygons left: 800 nsec
Adding a new edge: 133000 nsec
Checking if any edges remain: 3500 nsec
Calculating new edge parameters in ready_scan: 116363 nsec
Calculating new start_x and stop_x parameters: 31667 nsec
Calculating a new pixel: 600 nsec

Initializing a new fragment edge: 117600 nsec
Is edge horiz or vert?: 4334 nsec
Making a new fragment: 1000 nsec
Reading a dz value from the node buffer: 200 nsec
Insertion of a fragment into a frag list: 2000 nsec
Comparing the fragment depth: 650 nsec

Z buf read cycle: 200 nsec
Z buf write cycle: 200 nsec
Merging the last fragment: 16500 nsec
Merging other fragments: 24000 nsec
Traversing one link in a frag list: 400 nsec
Merging common ID fragments: 4667 nsec
Time to perform an if test and shift: 3500 nsec
Time to do a depth calc: 7333 nsec
Exchanging fragments on a list: 11000 nsec

Collection start time: 10000 nsec
Collection intervals: 1333 nsec
Clock start time: 0 nsec
Arbitrating a request without overlap: 50 nsec

Data Chararteristics:

Number of Polygons Rendered: 1585
Number of Transparent Polygons Rendered: 0
Number of Edges Rendered: 6116
Total Number of Pixels in Frame: 65536
Total Number of Pixels Covered: 14170
Total Number of Whole Pixels Generated: 13561
Total Number of Whole Pixels Stored: 10786 (includes overlaps)
Total Number of Whole Pixels On the Screen: 9339

Total Number of Fragments Generated: 38646
Total Number of Fragments Stored: 31047
Total Number of Antialiased Pixels on Screen: 6571

Percentage of Frame Covered: 21.6217
Whole Pixels (% of Frame): 14.2502
Aliased Pixels (All) (% of Frame): 11.8820
Aliased Pixels (Exposed) (% of Frame): 10.0266
Aliased Pixels (% of Stored Pixels): 46.3726
Average Number of Pixels per Edge: 6.3188
(if there are no transparent polygons)
Average Number of Pixels per Polygon: 8.9401
Average Number of Edges per Polygon: 3.8587
Average Number of Overlaps per Pixel: 2.6843

Maximum Number of Overlaps: 6
Maximum Number of Whole Pixel Stores: 5
Maximum Number of Fragment Overlaps: 103
Maximum Number of Fragment Stores: 85

SIMARCH PERFORMANCE OUTPUT FILE-run terminated on Wed Aug 20 14:09:47 1986
##